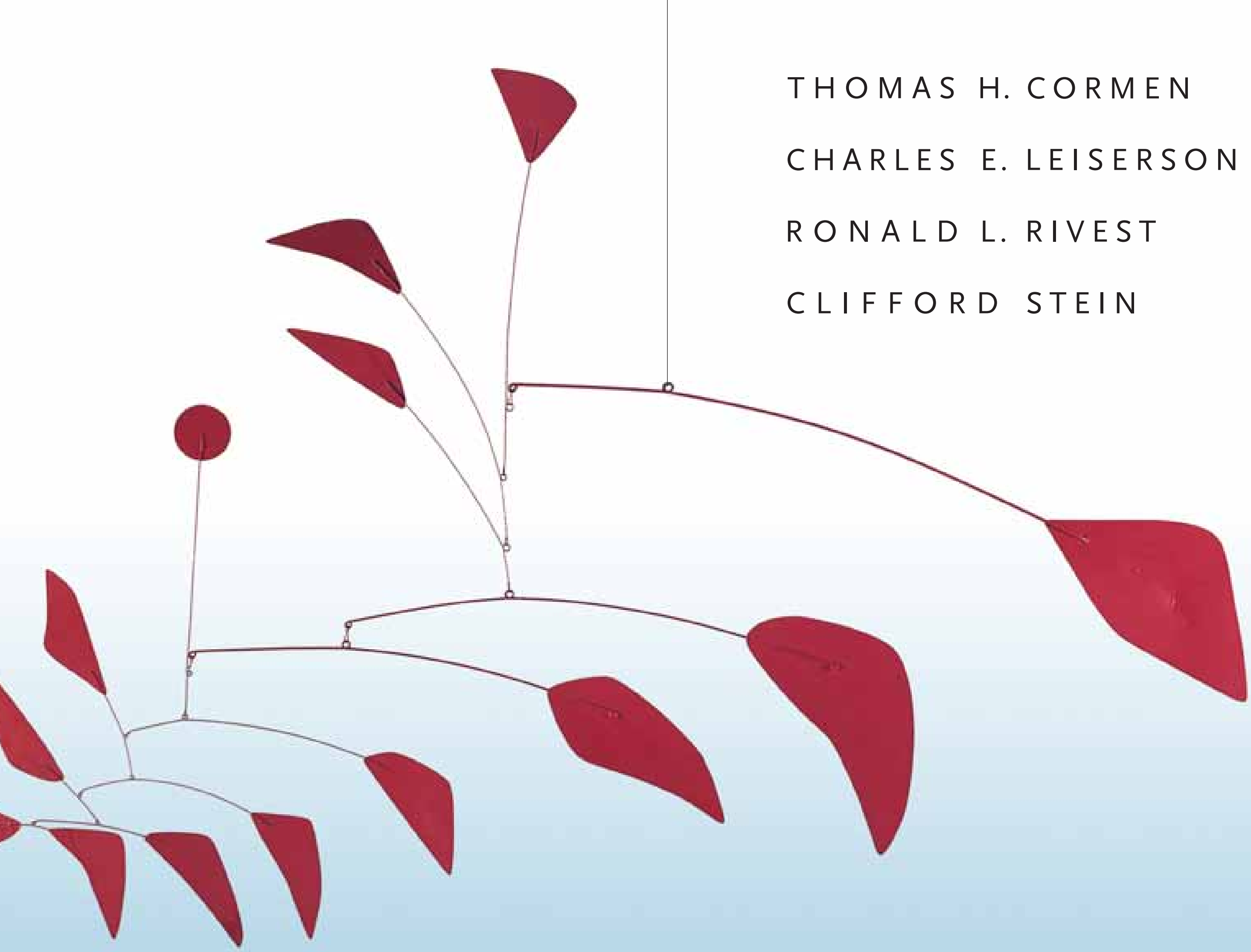
THOMAS H. CORMEN

CHARLES E. LEISERSON

RONALD L. RIVEST

CLIFFORD STEIN



INTRODUCTION TO ALGORITHMS

THIRD EDITION

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

Introduction to Algorithms

Third Edition

The MIT Press
Cambridge, Massachusetts London, England

This book was set in Times Roman and Mathtime Pro 2 by the authors.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Introduction to algorithms / Thomas H. Cormen . . . [et al.].—3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-03384-8 (hardcover : alk. paper)—ISBN 978-0-262-53305-8 (pbk. : alk. paper)
1. Computer programming. 2. Computer algorithms. I. Cormen, Thomas H.

QA76.6.I5858 2009
005.1—dc22
2009008593

10 9 8 7 6 5 4 3 2

Contents

II Sorting and Order Statistics

III Data Structures

VIII Appendix: Mathematical Background

Preface

Before there were computers, there were algorithms. But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing.

This book provides a comprehensive introduction to the modern study of computer algorithms. It presents many algorithms and covers them in considerable depth, yet makes their design and analysis accessible to all levels of readers. We have tried to keep explanations elementary without sacrificing depth of coverage or mathematical rigor.

Each chapter presents an algorithm, a design technique, an application area, or a related topic. Algorithms are described in English and in a pseudocode designed to be readable by anyone who has done a little programming. The book contains 244 figures—many with multiple parts—illustrating how the algorithms work. Since we emphasize *efficiency* as a design criterion, we include careful analyses of the running times of all our algorithms.

The text is intended primarily for use in undergraduate or graduate courses in algorithms or data structures. Because it discusses engineering issues in algorithm design, as well as mathematical aspects, it is equally well suited for self-study by technical professionals.

In this, the third edition, we have once again updated the entire book. The changes cover a broad spectrum, including new chapters, revised pseudocode, and a more active writing style.

To the teacher

We have designed this book to be both versatile and complete. You should find it useful for a variety of courses, from an undergraduate course in data structures up through a graduate course in algorithms. Because we have provided considerably more material than can fit in a typical one-term course, you can consider this book to be a "buffet" or "smorgasbord" from which you can pick and choose the material that best supports the course you wish to teach.

You should find it easy to organize your course around just the chapters you need. We have made chapters relatively self-contained, so that you need not worry about an unexpected and unnecessary dependence of one chapter on another. Each chapter presents the easier material first and the more difficult material later, with section boundaries marking natural stopping points. In an undergraduate course, you might use only the earlier sections from a chapter; in a graduate course, you might cover the entire chapter.

We have included 957 exercises and 158 problems. Each section ends with exercises, and each chapter ends with problems. The exercises are generally short questions that test basic mastery of the material. Some are simple self-check thought exercises, whereas others are more substantial and are suitable as assigned homework. The problems are more elaborate case studies that often introduce new material; they often consist of several questions that lead the student through the steps required to arrive at a solution.

Departing from our practice in previous editions of this book, we have made publicly available solutions to some, but by no means all, of the problems and exercises. Our Web site, http://mitpress.mit.edu/algorithms/, links to these solutions. You will want to check this site to make sure that it does not contain the solution to an exercise or problem that you plan to assign. We expect the set of solutions that we post to grow slowly over time, so you will need to check it each time you teach the course.

We have starred ($\star$) the sections and exercises that are more suitable for graduate students than for undergraduates. A starred section is not necessarily more difficult than an unstarred one, but it may require an understanding of more advanced mathematics. Likewise, starred exercises may require an advanced background or more than average creativity.

To the student

We hope that this textbook provides you with an enjoyable introduction to the field of algorithms. We have attempted to make every algorithm accessible and interesting. To help you when you encounter unfamiliar or difficult algorithms, we describe each one in a step-by-step manner. We also provide careful explanations of the mathematics needed to understand the analysis of the algorithms. If you already have some familiarity with a topic, you will find the chapters organized so that you can skim introductory sections and proceed quickly to the more advanced material.

This is a large book, and your class will probably cover only a portion of its material. We have tried, however, to make this a book that will be useful to you now as a course textbook and also later in your career as a mathematical desk reference or an engineering handbook.

What are the prerequisites for reading this book?

- You should have some programming experience. In particular, you should understand recursive procedures and simple data structures such as arrays and linked lists.
- You should have some facility with mathematical proofs, and especially proofs by mathematical induction. A few portions of the book rely on some knowledge of elementary calculus. Beyond that, Parts I and VIII of this book teach you all the mathematical techniques you will need.

We have heard, loud and clear, the call to supply solutions to problems and exercises. Our Web site, http://mitpress.mit.edu/algorithms/, links to solutions for a few of the problems and exercises. Feel free to check your solutions against ours. We ask, however, that you do not send your solutions to us.

To the professional

The wide range of topics in this book makes it an excellent handbook on algorithms. Because each chapter is relatively self-contained, you can focus in on the topics that most interest you.

Most of the algorithms we discuss have great practical utility. We therefore address implementation concerns and other engineering issues. We often provide practical alternatives to the few algorithms that are primarily of theoretical interest.

If you wish to implement any of the algorithms, you should find the translation of our pseudocode into your favorite programming language to be a fairly straightforward task. We have designed the pseudocode to present each algorithm clearly and succinctly. Consequently, we do not address error-handling and other software-engineering issues that require specific assumptions about your programming environment. We attempt to present each algorithm simply and directly without allowing the idiosyncrasies of a particular programming language to obscure its essence.

We understand that if you are using this book outside of a course, then you might be unable to check your solutions to problems and exercises against solutions provided by an instructor. Our Web site, http://mitpress.mit.edu/algorithms/, links to solutions for some of the problems and exercises so that you can check your work. Please do not send your solutions to us.

To our colleagues

We have supplied an extensive bibliography and pointers to the current literature. Each chapter ends with a set of chapter notes that give historical details and references. The chapter notes do not provide a complete reference to the whole field of algorithms, however. Though it may be hard to believe for a book of this size, space constraints prevented us from including many interesting algorithms.

Despite myriad requests from students for solutions to problems and exercises, we have chosen as a matter of policy not to supply references for problems and exercises, to remove the temptation for students to look up a solution rather than to find it themselves.

Changes for the third edition

What has changed between the second and third editions of this book? The magnitude of the changes is on a par with the changes between the first and second editions. As we said about the second-edition changes, depending on how you look at it, the book changed either not much or quite a bit.

A quick look at the table of contents shows that most of the second-edition chapters and sections appear in the third edition. We removed two chapters and one section, but we have added three new chapters and two new sections apart from these new chapters.

We kept the hybrid organization from the first two editions. Rather than organizing chapters by only problem domains or according only to techniques, this book has elements of both. It contains technique-based chapters on divide-and-conquer, dynamic programming, greedy algorithms, amortized analysis, NP-Completeness, and approximation algorithms. But it also has entire parts on sorting, on data structures for dynamic sets, and on algorithms for graph problems. We find that although you need to know how to apply techniques for designing and analyzing algorithms, problems seldom announce to you which techniques are most amenable to solving them.

Here is a summary of the most significant changes for the third edition:

- We added new chapters on van Emde Boas trees and multithreaded algorithms, and we have broken out material on matrix basics into its own appendix chapter.
- We revised the chapter on recurrences to more broadly cover the divide-and-conquer technique, and its first two sections apply divide-and-conquer to solve two problems. The second section of this chapter presents Strassen's algorithm for matrix multiplication, which we have moved from the chapter on matrix operations.
- We removed two chapters that were rarely taught: binomial heaps and sorting networks. One key idea in the sorting networks chapter, the 0-1 principle, appears in this edition within Problem 8-7 as the 0-1 sorting lemma for compare-exchange algorithms. The treatment of Fibonacci heaps no longer relies on binomial heaps as a precursor.

- We revised our treatment of dynamic programming and greedy algorithms. Dynamic programming now leads off with a more interesting problem, rod cutting, than the assembly-line scheduling problem from the second edition. Furthermore, we emphasize memoization a bit more than we did in the second edition, and we introduce the notion of the subproblem graph as a way to understand the running time of a dynamic-programming algorithm. In our opening example of greedy algorithms, the activity-selection problem, we get to the greedy algorithm more directly than we did in the second edition.
- The way we delete a node from binary search trees (which includes red-black trees) now guarantees that the node requested for deletion is the node that is actually deleted. In the first two editions, in certain cases, some other node would be deleted, with its contents moving into the node passed to the deletion procedure. With our new way to delete nodes, if other components of a program maintain pointers to nodes in the tree, they will not mistakenly end up with stale pointers to nodes that have been deleted.
- The material on flow networks now bases flows entirely on edges. This approach is more intuitive than the net flow used in the first two editions.
- With the material on matrix basics and Strassen's algorithm moved to other chapters, the chapter on matrix operations is smaller than in the second edition.
- We have modified our treatment of the Knuth-Morris-Pratt string-matching algorithm.
- We corrected several errors. Most of these errors were posted on our Web site of second-edition errata, but a few were not.
- Based on many requests, we changed the syntax (as it were) of our pseudocode. We now use "$=$" to indicate assignment and "==" to test for equality, just as C, C++, Java, and Python do. Likewise, we have eliminated the keywords **do** and **then** and adopted "**//**" as our comment-to-end-of-line symbol. We also now use dot-notation to indicate object attributes. Our pseudocode remains procedural, rather than object-oriented. In other words, rather than running methods on objects, we simply call procedures, passing objects as parameters.
- We added 100 new exercises and 28 new problems. We also updated many bibliography entries and added several new ones.
- Finally, we went through the entire book and rewrote sentences, paragraphs, and sections to make the writing clearer and more active.

Web site

You can use our Web site, http://mitpress.mit.edu/algorithms/, to obtain supplementary information and to communicate with us. The Web site links to a list of known errors, solutions to selected exercises and problems, and (of course) a list explaining the corny professor jokes, as well as other content that we might add. The Web site also tells you how to report errors or make suggestions.

How we produced this book

Like the second edition, the third edition was produced in $\LaTeX\,2_\varepsilon$. We used the Times font with mathematics typeset using the MathTime Pro 2 fonts. We thank Michael Spivak from Publish or Perish, Inc., Lance Carnes from Personal TeX, Inc., and Tim Tregubov from Dartmouth College for technical support. As in the previous two editions, we compiled the index using Windex, a C program that we wrote, and the bibliography was produced with BIBTEX. The PDF files for this book were created on a MacBook running OS 10.5.

We drew the illustrations for the third edition using MacDraw Pro, with some of the mathematical expressions in illustrations laid in with the psfrag package for $\LaTeX\,2_\varepsilon$. Unfortunately, MacDraw Pro is legacy software, having not been marketed for over a decade now. Happily, we still have a couple of Macintoshes that can run the Classic environment under OS 10.4, and hence they can run MacDraw Pro—mostly. Even under the Classic environment, we find MacDraw Pro to be far easier to use than any other drawing software for the types of illustrations that accompany computer-science text, and it produces beautiful output.[1] Who knows how long our pre-Intel Macs will continue to run, so if anyone from Apple is listening: *Please create an OS X-compatible version of MacDraw Pro!*

Acknowledgments for the third edition

We have been working with the MIT Press for over two decades now, and what a terrific relationship it has been! We thank Ellen Faran, Bob Prior, Ada Brunstein, and Mary Reilly for their help and support.

We were geographically distributed while producing the third edition, working in the Dartmouth College Department of Computer Science, the MIT Computer

[1] We investigated several drawing programs that run under Mac OS X, but all had significant shortcomings compared with MacDraw Pro. We briefly attempted to produce the illustrations for this book with a different, well known drawing program. We found that it took at least five times as long to produce each illustration as it took with MacDraw Pro, and the resulting illustrations did not look as good. Hence the decision to revert to MacDraw Pro running on older Macintoshes.

Science and Artificial Intelligence Laboratory, and the Columbia University Department of Industrial Engineering and Operations Research. We thank our respective universities and colleagues for providing such supportive and stimulating environments.

Julie Sussman, P.P.A., once again bailed us out as the technical copyeditor. Time and again, we were amazed at the errors that eluded us, but that Julie caught. She also helped us improve our presentation in several places. If there is a Hall of Fame for technical copyeditors, Julie is a sure-fire, first-ballot inductee. She is nothing short of phenomenal. Thank you, thank you, thank you, Julie! Priya Natarajan also found some errors that we were able to correct before this book went to press. Any errors that remain (and undoubtedly, some do) are the responsibility of the authors (and probably were inserted after Julie read the material).

The treatment for van Emde Boas trees derives from Erik Demaine's notes, which were in turn influenced by Michael Bender. We also incorporated ideas from Javed Aslam, Bradley Kuszmaul, and Hui Zha into this edition.

The chapter on multithreading was based on notes originally written jointly with Harald Prokop. The material was influenced by several others working on the Cilk project at MIT, including Bradley Kuszmaul and Matteo Frigo. The design of the multithreaded pseudocode took its inspiration from the MIT Cilk extensions to C and by Cilk Arts's Cilk++ extensions to C++.

We also thank the many readers of the first and second editions who reported errors or submitted suggestions for how to improve this book. We corrected all the bona fide errors that were reported, and we incorporated as many suggestions as we could. We rejoice that the number of such contributors has grown so great that we must regret that it has become impractical to list them all.

Finally, we thank our wives—Nicole Cormen, Wendy Leiserson, Gail Rivest, and Rebecca Ivry—and our children—Ricky, Will, Debby, and Katie Leiserson; Alex and Christopher Rivest; and Molly, Noah, and Benjamin Stein—for their love and support while we prepared this book. The patience and encouragement of our families made this project possible. We affectionately dedicate this book to them.

THOMAS H. CORMEN	*Lebanon, New Hampshire*
CHARLES E. LEISERSON	*Cambridge, Massachusetts*
RONALD L. RIVEST	*Cambridge, Massachusetts*
CLIFFORD STEIN	*New York, New York*

February 2009

Divide and Conquer

4 Divide-and-Conquer

In Section 2.3.1, we saw how merge sort serves as an example of the divide-and-conquer paradigm. Recall that in divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough to solve recursively, we call that the ***recursive case***. Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the ***base case***. Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem. We consider solving such subproblems as part of the combine step.

In this chapter, we shall see more algorithms based on divide-and-conquer. The first one solves the maximum-subarray problem: it takes as input an array of numbers, and it determines the contiguous subarray whose values have the greatest sum. Then we shall see two divide-and-conquer algorithms for multiplying $n \times n$ matrices. One runs in $\Theta(n^3)$ time, which is no better than the straightforward method of multiplying square matrices. But the other, Strassen's algorithm, runs in $O(n^{2.81})$ time, which beats the straightforward method asymptotically.

Recurrences

Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running times of divide-and-conquer algorithms. A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, in Section 2.3.2 we described the worst-case running time $T(n)$ of the MERGE-SORT procedure by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 , \end{cases} \tag{4.1}$$

whose solution we claimed to be $T(n) = \Theta(n \lg n)$.

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a $2/3$-to-$1/3$ split. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

Subproblems are not necessarily constrained to being a constant fraction of the original problem size. For example, a recursive version of linear search (see Exercise 2.1-3) would create just one subproblem containing only one element fewer than the original problem. Each recursive call would take constant time plus the time for the recursive calls it makes, yielding the recurrence $T(n) = T(n-1) + \Theta(1)$.

This chapter offers three methods for solving recurrences—that is, for obtaining asymptotic "Θ" or "O" bounds on the solution:

- In the ***substitution method***, we guess a bound and then use mathematical induction to prove our guess correct.
- The ***recursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- The ***master method*** provides bounds for recurrences of the form

  $$T(n) = aT(n/b) + f(n) , \tag{4.2}$$

  where $a \geq 1$, $b > 1$, and $f(n)$ is a given function. Such recurrences arise frequently. A recurrence of the form in equation (4.2) characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time.

  To use the master method, you will need to memorize three cases, but once you do that, you will easily be able to determine asymptotic bounds for many simple recurrences. We will use the master method to determine the running times of the divide-and-conquer algorithms for the maximum-subarray problem and for matrix multiplication, as well as for other algorithms based on divide-and-conquer elsewhere in this book.

Occasionally, we shall see recurrences that are not equalities but rather inequalities, such as $T(n) \le 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we will couch its solution using O-notation rather than Θ-notation. Similarly, if the inequality were reversed to $T(n) \ge 2T(n/2) + \Theta(n)$, then because the recurrence gives only a lower bound on $T(n)$, we would use Ω-notation in its solution.

Technicalities in recurrences

In practice, we neglect certain technical details when we state and solve recurrences. For example, if we call MERGE-SORT on n elements when n is odd, we end up with subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Neither size is actually $n/2$, because $n/2$ is not an integer when n is odd. Technically, the recurrence describing the worst-case running time of MERGE-SORT is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{4.3}$$

Boundary conditions represent another class of details that we typically ignore. Since the running time of an algorithm on a constant-sized input is a constant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. Consequently, for convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n. For example, we normally state recurrence (4.1) as

$$T(n) = 2T(n/2) + \Theta(n) , \tag{4.4}$$

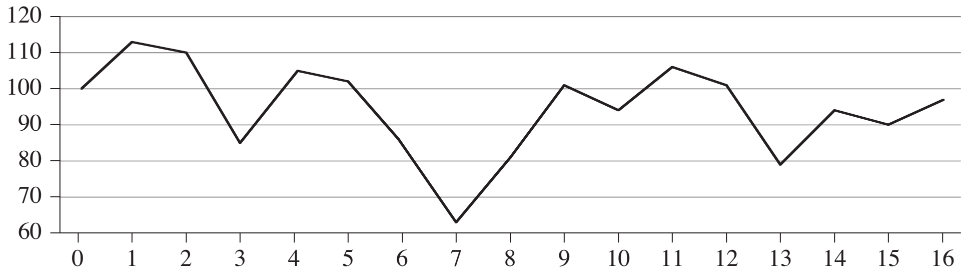
without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions. We forge ahead without these details and later determine whether or not they matter. They usually do not, but you should know when they do. Experience helps, and so do some theorems stating that these details do not affect the asymptotic bounds of many recurrences characterizing divide-and-conquer algorithms (see Theorem 4.1). In this chapter, however, we shall address some of these details and illustrate the fine points of recurrence solution methods.

4.1 The maximum-subarray problem

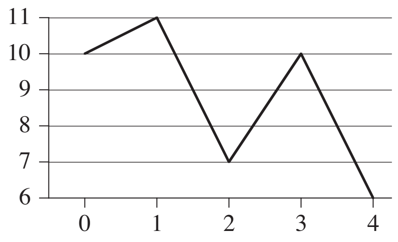
Suppose that you been offered the opportunity to invest in the Volatile Chemical Corporation. Like the chemicals the company produces, the stock price of the Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day. To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future. Your goal is to maximize your profit. Figure 4.1 shows the price of the stock over a 17-day period. You may buy the stock at any one time, starting after day 0, when the price is \$100 per share. Of course, you would want to "buy low, sell high"—buy at the lowest possible price and later on sell at the highest possible price—to maximize your profit. Unfortunately, you might not be able to buy at the lowest price and then sell at the highest price within a given period. In Figure 4.1, the lowest price occurs after day 7, which occurs after the highest price, after day 1.

You might think that you can always maximize profit by either buying at the lowest price or selling at the highest price. For example, in Figure 4.1, we would maximize profit by buying at the lowest price, after day 7. If this strategy always worked, then it would be easy to determine how to maximize profit: find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference. Figure 4.2 shows a simple counterexample,



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

demonstrating that the maximum profit sometimes comes neither by buying at the lowest price nor by selling at the highest price.

A brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. A period of n days has $\binom{n}{2}$ such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time. Can we do better?

A transformation

In order to design an algorithm with an $o(n^2)$ running time, we will look at the input in a slightly different way. We want to find a sequence of days over which the net change from the first day to the last is maximum. Instead of looking at the daily prices, let us instead consider the daily change in price, where the change on day i is the difference between the prices after day $i - 1$ and after day i. The table in Figure 4.1 shows these daily changes in the bottom row. If we treat this row as an array A, shown in Figure 4.3, we now want to find the nonempty, contiguous subarray of A whose values have the largest sum. We call this contiguous subarray the ***maximum subarray***. For example, in the array of Figure 4.3, the maximum subarray of $A[1..16]$ is $A[8..11]$, with the sum 43. Thus, you would want to buy the stock just before day 8 (that is, after day 7) and sell it after day 11, earning a profit of \$43 per share.

At first glance, this transformation does not help. We still need to check $\binom{n-1}{2} = \Theta(n^2)$ subarrays for a period of n days. Exercise 4.1-2 asks you to show

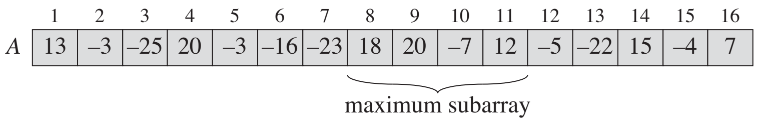


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8\,..\,11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

that although computing the cost of one subarray might take time proportional to the length of the subarray, when computing all $\Theta(n^2)$ subarray sums, we can organize the computation so that each subarray sum takes $O(1)$ time, given the values of previously computed subarray sums, so that the brute-force solution takes $\Theta(n^2)$ time.

So let us seek a more efficient solution to the maximum-subarray problem. When doing so, we will usually speak of "a" maximum subarray rather than "the" maximum subarray, since there could be more than one subarray that achieves the maximum sum.

The maximum-subarray problem is interesting only when the array contains some negative numbers. If all the array entries were nonnegative, then the maximum-subarray problem would present no challenge, since the entire array would give the greatest sum.

A solution using divide-and-conquer

Let's think about how we might solve the maximum-subarray problem using the divide-and-conquer technique. Suppose we want to find a maximum subarray of the subarray $A[low\,..\,high]$. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say *mid*, of the subarray, and consider the subarrays $A[low\,..\,mid]$ and $A[mid+1\,..\,high]$. As Figure 4.4(a) shows, any contiguous subarray $A[i\,..\,j]$ of $A[low\,..\,high]$ must lie in exactly one of the following places:

- entirely in the subarray $A[low\,..\,mid]$, so that $low \le i \le j \le mid$,
- entirely in the subarray $A[mid+1\,..\,high]$, so that $mid < i \le j \le high$, or
- crossing the midpoint, so that $low \le i \le mid < j \le high$.

Therefore, a maximum subarray of $A[low\,..\,high]$ must lie in exactly one of these places. In fact, a maximum subarray of $A[low\,..\,high]$ must have the greatest sum over all subarrays entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint. We can find maximum subarrays of $A[low\,..\,mid]$ and $A[mid+1\,..\,high]$ recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray. Thus, all that is left to do is find a

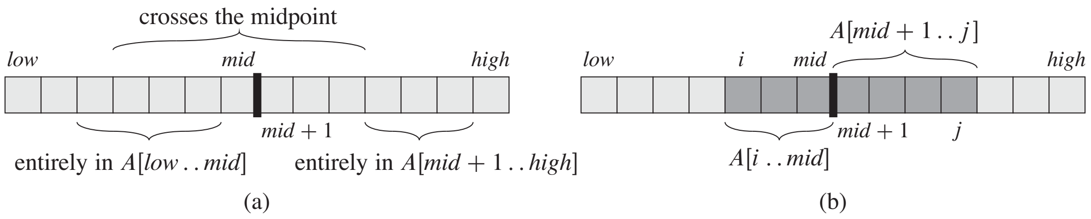


Figure 4.4 **(a)** Possible locations of subarrays of $A[low \,..\, high]$: entirely in $A[low \,..\, mid]$, entirely in $A[mid + 1 \,..\, high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low \,..\, high]$ crossing the midpoint comprises two subarrays $A[i \,..\, mid]$ and $A[mid + 1 \,..\, j]$, where $low \le i \le mid$ and $mid < j \le high$.

maximum subarray that crosses the midpoint, and take a subarray with the largest sum of the three.

We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray $A[low \,..\, high]$. This problem is *not* a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint. As Figure 4.4(b) shows, any subarray crossing the midpoint is itself made of two subarrays $A[i \,..\, mid]$ and $A[mid + 1 \,..\, j]$, where $low \le i \le mid$ and $mid < j \le high$. Therefore, we just need to find maximum subarrays of the form $A[i \,..\, mid]$ and $A[mid + 1 \,..\, j]$ and then combine them. The procedure FIND-MAX-CROSSING-SUBARRAY takes as input the array A and the indices *low*, *mid*, and *high*, and it returns a tuple containing the indices demarcating a maximum subarray that crosses the midpoint, along with the sum of the values in a maximum subarray.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4      sum = sum + A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

This procedure works as follows. Lines 1–7 find a maximum subarray of the left half, $A[low \, .\, . \, mid]$. Since this subarray must contain $A[mid]$, the **for** loop of lines 3–7 starts the index i at *mid* and works down to *low*, so that every subarray it considers is of the form $A[i \, .\, . \, mid]$. Lines 1–2 initialize the variables *left-sum*, which holds the greatest sum found so far, and *sum*, holding the sum of the entries in $A[i \, .\, . \, mid]$. Whenever we find, in line 5, a subarray $A[i \, .\, . \, mid]$ with a sum of values greater than *left-sum*, we update *left-sum* to this subarray's sum in line 6, and in line 7 we update the variable *max-left* to record this index i. Lines 8–14 work analogously for the right half, $A[mid+1 \, .\, . \, high]$. Here, the **for** loop of lines 10–14 starts the index j at $mid+1$ and works up to *high*, so that every subarray it considers is of the form $A[mid + 1 \, .\, . \, j]$. Finally, line 15 returns the indices *max-left* and *max-right* that demarcate a maximum subarray crossing the midpoint, along with the sum $\mathit{left\text{-}sum}+\mathit{right\text{-}sum}$ of the values in the subarray $A[\mathit{max\text{-}left} \, .\, . \, \mathit{max\text{-}right}]$.

If the subarray $A[low \, .\, . \, high]$ contains n entries (so that $n = high - low + 1$), we claim that the call FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$ takes $\Theta(n)$ time. Since each iteration of each of the two **for** loops takes $\Theta(1)$ time, we just need to count up how many iterations there are altogether. The **for** loop of lines 3–7 makes $mid - low + 1$ iterations, and the **for** loop of lines 10–14 makes $high - mid$ iterations, and so the total number of iterations is

$$\begin{aligned}(mid - low + 1) + (high - mid) &= high - low + 1 \\ &= n \,.\end{aligned}$$

With a linear-time FIND-MAX-CROSSING-SUBARRAY procedure in hand, we can write pseudocode for a divide-and-conquer algorithm to solve the maximum-subarray problem:

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

The initial call FIND-MAXIMUM-SUBARRAY$(A, 1, A.length)$ will find a maximum subarray of $A[1 \mathinner{..} n]$.

Similar to FIND-MAX-CROSSING-SUBARRAY, the recursive procedure FIND-MAXIMUM-SUBARRAY returns a tuple containing the indices that demarcate a maximum subarray, along with the sum of the values in a maximum subarray. Line 1 tests for the base case, where the subarray has just one element. A subarray with just one element has only one subarray—itself—and so line 2 returns a tuple with the starting and ending indices of just the one element, along with its value. Lines 3–11 handle the recursive case. Line 3 does the divide part, computing the index *mid* of the midpoint. Let's refer to the subarray $A[low \mathinner{..} mid]$ as the ***left subarray*** and to $A[mid + 1 \mathinner{..} high]$ as the ***right subarray***. Because we know that the subarray $A[low \mathinner{..} high]$ contains at least two elements, each of the left and right subarrays must have at least one element. Lines 4 and 5 conquer by recursively finding maximum subarrays within the left and right subarrays, respectively. Lines 6–11 form the combine part. Line 6 finds a maximum subarray that crosses the midpoint. (Recall that because line 6 solves a subproblem that is not a smaller instance of the original problem, we consider it to be in the combine part.) Line 7 tests whether the left subarray contains a subarray with the maximum sum, and line 8 returns that maximum subarray. Otherwise, line 9 tests whether the right subarray contains a subarray with the maximum sum, and line 10 returns that maximum subarray. If neither the left nor right subarrays contain a subarray achieving the maximum sum, then a maximum subarray must cross the midpoint, and line 11 returns it.

Analyzing the divide-and-conquer algorithm

Next we set up a recurrence that describes the running time of the recursive FIND-MAXIMUM-SUBARRAY procedure. As we did when we analyzed merge sort in Section 2.3.2, we make the simplifying assumption that the original problem size is a power of 2, so that all subproblem sizes are integers. We denote by $T(n)$ the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements. For starters, line 1 takes constant time. The base case, when $n = 1$, is easy: line 2 takes constant time, and so

$$T(1) = \Theta(1) . \tag{4.5}$$

The recursive case occurs when $n > 1$. Lines 1 and 3 take constant time. Each of the subproblems solved in lines 4 and 5 is on a subarray of $n/2$ elements (our assumption that the original problem size is a power of 2 ensures that $n/2$ is an integer), and so we spend $T(n/2)$ time solving each of them. Because we have to solve two subproblems—for the left subarray and for the right subarray—the contribution to the running time from lines 4 and 5 comes to $2T(n/2)$. As we have already seen, the call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes $\Theta(n)$ time. Lines 7–11 take only $\Theta(1)$ time. For the recursive case, therefore, we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) . \end{aligned} \tag{4.6}$$

Combining equations (4.5) and (4.6) gives us a recurrence for the running time $T(n)$ of FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{4.7}$$

This recurrence is the same as recurrence (4.1) for merge sort. As we shall see from the master method in Section 4.5, this recurrence has the solution $T(n) = \Theta(n \lg n)$. You might also revisit the recursion tree in Figure 2.5 to understand why the solution should be $T(n) = \Theta(n \lg n)$.

Thus, we see that the divide-and-conquer method yields an algorithm that is asymptotically faster than the brute-force method. With merge sort and now the maximum-subarray problem, we begin to get an idea of how powerful the divide-and-conquer method can be. Sometimes it will yield the asymptotically fastest algorithm for a problem, and other times we can do even better. As Exercise 4.1-5 shows, there is in fact a linear-time algorithm for the maximum-subarray problem, and it does not use divide-and-conquer.

Exercises

4.1-1
What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?

4.1-2
Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in $\Theta(n^2)$ time.

4.1-3
Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size n_0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n_0. Does that change the crossover point?

4.1-4
Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subar-

ray is 0. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?

4.1-5

Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1 \mathinner{..} j]$, extend the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray of $A[1 \mathinner{..} j+1]$ is either a maximum subarray of $A[1 \mathinner{..} j]$ or a subarray $A[i \mathinner{..} j+1]$, for some $1 \le i \le j+1$. Determine a maximum subarray of the form $A[i \mathinner{..} j+1]$ in constant time based on knowing a maximum subarray ending at index j.

4.2 Strassen's algorithm for matrix multiplication

If you have seen matrices before, then you probably know how to multiply them. (Otherwise, you should read Section D.1 in Appendix D.) If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A \cdot B$, we define the entry c_{ij}, for $i, j = 1, 2, \ldots, n$, by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ . \tag{4.8}$$

We must compute n^2 matrix entries, and each is the sum of n values. The following procedure takes $n \times n$ matrices A and B and multiplies them, returning their $n \times n$ product C. We assume that each matrix has an attribute *rows*, giving the number of rows in the matrix.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  $n = A.rows$
2  let $C$ be a new $n \times n$ matrix
3  **for** $i = 1$ **to** $n$
4      **for** $j = 1$ **to** $n$
5          $c_{ij} = 0$
6          **for** $k = 1$ **to** $n$
7              $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
8  **return** $C$
```

The SQUARE-MATRIX-MULTIPLY procedure works as follows. The **for** loop of lines 3–7 computes the entries of each row i, and within a given row i, the

for loop of lines 4–7 computes each of the entries c_{ij}, for each column j. Line 5 initializes c_{ij} to 0 as we start computing the sum given in equation (4.8), and each iteration of the **for** loop of lines 6–7 adds in one more term of equation (4.8).

Because each of the triply-nested **for** loops runs exactly n iterations, and each execution of line 7 takes constant time, the SQUARE-MATRIX-MULTIPLY procedure takes $\Theta(n^3)$ time.

You might at first think that any matrix multiplication algorithm must take $\Omega(n^3)$ time, since the natural definition of matrix multiplication requires that many multiplications. You would be incorrect, however: we have a way to multiply matrices in $o(n^3)$ time. In this section, we shall see Strassen's remarkable recursive algorithm for multiplying $n \times n$ matrices. It runs in $\Theta(n^{\lg 7})$ time, which we shall show in Section 4.5. Since $\lg 7$ lies between 2.80 and 2.81, Strassen's algorithm runs in $O(n^{2.81})$ time, which is asymptotically better than the simple SQUARE-MATRIX-MULTIPLY procedure.

A simple divide-and-conquer algorithm

To keep things simple, when we use a divide-and-conquer algorithm to compute the matrix product $C = A \cdot B$, we assume that n is an exact power of 2 in each of the $n \times n$ matrices. We make this assumption because in each divide step, we will divide $n \times n$ matrices into four $n/2 \times n/2$ matrices, and by assuming that n is an exact power of 2, we are guaranteed that as long as $n \geq 2$, the dimension $n/2$ is an integer.

Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm:

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1
 4      c_11 = a_11 · b_11
 5  else partition A, B, and C as in equations (4.9)
 6      C_11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_11, B_11)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_12, B_21)
 7      C_12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_11, B_12)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_12, B_22)
 8      C_21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_21, B_11)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_22, B_21)
 9      C_22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_21, B_12)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_22, B_22)
10  return C
```

This pseudocode glosses over one subtle but important implementation detail. How do we partition the matrices in line 5? If we were to create 12 new $n/2 \times n/2$ matrices, we would spend $\Theta(n^2)$ time copying entries. In fact, we can partition the matrices without copying entries. The trick is to use index calculations. We identify a submatrix by a range of row indices and a range of column indices of the original matrix. We end up representing a submatrix a little differently from how we represent the original matrix, which is the subtlety we are glossing over. The advantage is that, since we can specify submatrices by index calculations, executing line 5 takes only $\Theta(1)$ time (although we shall see that it makes no difference asymptotically to the overall running time whether we copy or partition in place).

Now, we derive a recurrence to characterize the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE. Let $T(n)$ be the time to multiply two $n \times n$ matrices using this procedure. In the base case, when $n = 1$, we perform just the one scalar multiplication in line 4, and so

$$T(1) = \Theta(1) . \tag{4.15}$$

The recursive case occurs when $n > 1$. As discussed, partitioning the matrices in line 5 takes $\Theta(1)$ time, using index calculations. In lines 6–9, we recursively call SQUARE-MATRIX-MULTIPLY-RECURSIVE a total of eight times. Because each recursive call multiplies two $n/2 \times n/2$ matrices, thereby contributing $T(n/2)$ to the overall running time, the time taken by all eight recursive calls is $8T(n/2)$. We also must account for the four matrix additions in lines 6–9. Each of these matrices contains $n^2/4$ entries, and so each of the four matrix additions takes $\Theta(n^2)$ time. Since the number of matrix additions is a constant, the total time spent adding matrices in lines 6–9 is $\Theta(n^2)$. (Again, we use index calculations to place the results of the matrix additions into the correct positions of matrix C, with an overhead of $\Theta(1)$ time per entry.) The total time for the recursive case, therefore, is the sum of the partitioning time, the time for all the recursive calls, and the time to add the matrices resulting from the recursive calls:

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2)\ . \end{aligned} \tag{4.16}$$

Notice that if we implemented partitioning by copying matrices, which would cost $\Theta(n^2)$ time, the recurrence would not change, and hence the overall running time would increase by only a constant factor.

Combining equations (4.15) and (4.16) gives us the recurrence for the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\ , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1\ . \end{cases} \tag{4.17}$$

As we shall see from the master method in Section 4.5, recurrence (4.17) has the solution $T(n) = \Theta(n^3)$. Thus, this simple divide-and-conquer approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure.

Before we continue on to examining Strassen's algorithm, let us review where the components of equation (4.16) came from. Partitioning each $n \times n$ matrix by index calculation takes $\Theta(1)$ time, but we have two matrices to partition. Although you could say that partitioning the two matrices takes $\Theta(2)$ time, the constant of 2 is subsumed by the Θ-notation. Adding two matrices, each with, say, k entries, takes $\Theta(k)$ time. Since the matrices we add each have $n^2/4$ entries, you could say that adding each pair takes $\Theta(n^2/4)$ time. Again, however, the Θ-notation subsumes the constant factor of $1/4$, and we say that adding two $n^2/4 \times n^2/4$ matrices takes $\Theta(n^2)$ time. We have four such matrix additions, and once again, instead of saying that they take $\Theta(4n^2)$ time, we say that they take $\Theta(n^2)$ time. (Of course, you might observe that we could say that the four matrix additions take $\Theta(4n^2/4)$ time, and that $4n^2/4 = n^2$, but the point here is that Θ-notation subsumes constant factors, whatever they are.) Thus, we end up with two terms of $\Theta(n^2)$, which we can combine into one.

When we account for the eight recursive calls, however, we cannot just subsume the constant factor of 8. In other words, we must say that together they take $8T(n/2)$ time, rather than just $T(n/2)$ time. You can get a feel for why by looking back at the recursion tree in Figure 2.5, for recurrence (2.1) (which is identical to recurrence (4.7)), with the recursive case $T(n) = 2T(n/2) + \Theta(n)$. The factor of 2 determined how many children each tree node had, which in turn determined how many terms contributed to the sum at each level of the tree. If we were to ignore the factor of 8 in equation (4.16) or the factor of 2 in recurrence (4.1), the recursion tree would just be linear, rather than "bushy," and each level would contribute only one term to the sum.

Bear in mind, therefore, that although asymptotic notation subsumes constant multiplicative factors, recursive notation such as $T(n/2)$ does not.

Strassen's method

The key to Strassen's method is to make the recursion tree slightly less bushy. That is, instead of performing eight recursive multiplications of $n/2 \times n/2$ matrices, it performs only seven. The cost of eliminating one matrix multiplication will be several new additions of $n/2 \times n/2$ matrices, but still only a constant number of additions. As before, the constant number of matrix additions will be subsumed by Θ-notation when we set up the recurrence equation to characterize the running time.

Strassen's method is not at all obvious. (This might be the biggest understatement in this book.) It has four steps:

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

We shall see the details of steps 2–4 in a moment, but we already have enough information to set up a recurrence for the running time of Strassen's method. Let us assume that once the matrix size n gets down to 1, we perform a simple scalar multiplication, just as in line 4 of SQUARE-MATRIX-MULTIPLY-RECURSIVE. When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of $n/2 \times n/2$ matrices. Hence, we obtain the following recurrence for the running time $T(n)$ of Strassen's algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases} \tag{4.18}$$

We have traded off one matrix multiplication for a constant number of matrix additions. Once we understand recurrences and their solutions, we shall see that this tradeoff actually leads to a lower asymptotic running time. By the master method in Section 4.5, recurrence (4.18) has the solution $T(n) = \Theta(n^{\lg 7})$.

We now proceed to describe the details. In step 2, we create the following 10 matrices:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}$$

Since we must add or subtract $n/2 \times n/2$ matrices 10 times, this step does indeed take $\Theta(n^2)$ time.

In step 3, we recursively multiply $n/2 \times n/2$ matrices seven times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}$$

Note that the only multiplications we need to perform are those in the middle column of the above equations. The right-hand column just shows what these products equal in terms of the original submatrices created in step 1.

Step 4 adds and subtracts the P_i matrices created in step 3 to construct the four $n/2 \times n/2$ submatrices of the product C. We start with

$$C_{11} = P_5 + P_4 - P_2 + P_6 .$$

Expanding out the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that C_{11} equals

$$\begin{array}{lllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & & \\
 & - A_{11} \cdot B_{22} & & & & - A_{12} \cdot B_{22} & \\
 & & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & & & & + A_{12} \cdot B_{21} \ ,
\end{array}$$

which corresponds to equation (4.11).

Similarly, we set

$$C_{12} = P_1 + P_2 \ ,$$

and so C_{12} equals

$$\begin{array}{lll}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} & \\
 & + A_{11} \cdot B_{22} & + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22} \ ,
\end{array}$$

corresponding to equation (4.12).

Setting

$$C_{21} = P_3 + P_4$$

makes C_{21} equal

$$\begin{array}{lll}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} & \\
 & - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21} \ ,
\end{array}$$

corresponding to equation (4.13).

Finally, we set

$$C_{22} = P_5 + P_1 - P_3 - P_7 \ ,$$

so that C_{22} equals

$$\begin{array}{lllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12} \ ,
\end{array}$$

which corresponds to equation (4.14). Altogether, we add or subtract $n/2 \times n/2$ matrices eight times in step 4, and so this step indeed takes $\Theta(n^2)$ time.

Thus, we see that Strassen's algorithm, comprising steps 1–4, produces the correct matrix product and that recurrence (4.18) characterizes its running time. Since we shall see in Section 4.5 that this recurrence has the solution $T(n) = \Theta(n^{\lg 7})$, Strassen's method is asymptotically faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure. The notes at the end of this chapter discuss some of the practical aspects of Strassen's algorithm.

Exercises

Note: Although Exercises 4.2-3, 4.2-4, and 4.2-5 are about variants on Strassen's algorithm, you should read Section 4.5 before trying to solve them.

4.2-1
Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix} .$$

Show your work.

4.2-2
Write pseudocode for Strassen's algorithm.

4.2-3
How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.

4.2-4
What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

4.2-5
V. Pan has discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare to Strassen's algorithm?

4.2-6
How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.

4.2-7
Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

4.3 The substitution method for solving recurrences

Now that we have seen how recurrences characterize the running times of divide-and-conquer algorithms, we will learn how to solve recurrences. We start in this section with the "substitution" method.

The ***substitution method*** for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name "substitution method." This method is powerful, but we must be able to guess the form of the answer in order to apply it.

We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \ , \tag{4.19}$$

which is similar to recurrences (4.3) and (4.4). We guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \le cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n \ , \end{aligned}$$

where the last step holds as long as $c \geq 1$.

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. For the recurrence (4.19), we must show that we can choose the constant c large enough so that the bound $T(n) \leq cn \lg n$ works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that $T(1) = 1$ is the sole boundary condition of the recurrence. Then for $n = 1$, the bound $T(n) \leq cn \lg n$ yields $T(1) \leq c1 \lg 1 = 0$, which is at odds with $T(1) = 1$. Consequently, the base case of our inductive proof fails to hold.

We can overcome this obstacle in proving an inductive hypothesis for a specific boundary condition with only a little more effort. In the recurrence (4.19), for example, we take advantage of asymptotic notation requiring us only to prove $T(n) \leq cn \lg n$ for $n \geq n_0$, where n_0 is a constant *that we get to choose*. We keep the troublesome boundary condition $T(1) = 1$, but remove it from consideration in the inductive proof. We do so by first observing that for $n > 3$, the recurrence does not depend directly on $T(1)$. Thus, we can replace $T(1)$ by $T(2)$ and $T(3)$ as the base cases in the inductive proof, letting $n_0 = 2$. Note that we make a distinction between the base case of the recurrence ($n = 1$) and the base cases of the inductive proof ($n = 2$ and $n = 3$). With $T(1) = 1$, we derive from the recurrence that $T(2) = 4$ and $T(3) = 5$. Now we can complete the inductive proof that $T(n) \leq cn \lg n$ for some constant $c \geq 1$ by choosing c large enough so that $T(2) \leq c2 \lg 2$ and $T(3) \leq c3 \lg 3$. As it turns out, any choice of $c \geq 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold. For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n, and we shall not always explicitly work out the details.

Making a good guess

Unfortunately, there is no general way to guess the correct solutions to recurrences. Guessing a solution takes experience and, occasionally, creativity. Fortunately, though, you can use some heuristics to help you become a good guesser. You can also use recursion trees, which we shall see in Section 4.4, to generate good guesses.

If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable. As an example, consider the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n ,$$

which looks difficult because of the added "17" in the argument to T on the right-hand side. Intuitively, however, this additional term cannot substantially affect the solution to the recurrence. When n is large, the difference between $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 17$ is not that large: both cut n nearly evenly in half. Consequently, we make the guess that $T(n) = O(n \lg n)$, which you can verify as correct by using the substitution method (see Exercise 4.3-6).

Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty. For example, we might start with a lower bound of $T(n) = \Omega(n)$ for the recurrence (4.19), since we have the term n in the recurrence, and we can prove an initial upper bound of $T(n) = O(n^2)$. Then, we can gradually lower the upper bound and raise the lower bound until we converge on the correct, asymptotically tight solution of $T(n) = \Theta(n \lg n)$.

Subtleties

Sometimes you might correctly guess an asymptotic bound on the solution of a recurrence, but somehow the math fails to work out in the induction. The problem frequently turns out to be that the inductive assumption is not strong enough to prove the detailed bound. If you revise the guess by subtracting a lower-order term when you hit such a snag, the math often goes through.

Consider the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 .$$

We guess that the solution is $T(n) = O(n)$, and we try to show that $T(n) \le cn$ for an appropriate choice of the constant c. Substituting our guess in the recurrence, we obtain

$$\begin{aligned} T(n) &\le c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &= cn + 1 , \end{aligned}$$

which does not imply $T(n) \le cn$ for any choice of c. We might be tempted to try a larger guess, say $T(n) = O(n^2)$. Although we can make this larger guess work, our original guess of $T(n) = O(n)$ is correct. In order to show that it is correct, however, we must make a stronger inductive hypothesis.

Intuitively, our guess is nearly right: we are off only by the constant 1, a lower-order term. Nevertheless, mathematical induction does not work unless we prove the exact form of the inductive hypothesis. We overcome our difficulty by *subtracting* a lower-order term from our previous guess. Our new guess is $T(n) \le cn - d$, where $d \ge 0$ is a constant. We now have

$$\begin{aligned} T(n) &\le (c \lfloor n/2 \rfloor - d) + (c \lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\le cn - d , \end{aligned}$$

as long as $d \geq 1$. As before, we must choose the constant c large enough to handle the boundary conditions.

You might find the idea of subtracting a lower-order term counterintuitive. After all, if the math does not work out, we should increase our guess, right? Not necessarily! When proving an upper bound by induction, it may actually be more difficult to prove that a weaker upper bound holds, because in order to prove the weaker bound, we must use the same weaker bound inductively in the proof. In our current example, when the recurrence has more than one recursive term, we get to subtract out the lower-order term of the proposed bound once per recursive term. In the above example, we subtracted out the constant d twice, once for the $T(\lfloor n/2 \rfloor)$ term and once for the $T(\lceil n/2 \rceil)$ term. We ended up with the inequality $T(n) \leq cn - 2d + 1$, and it was easy to find values of d to make $cn - 2d + 1$ be less than or equal to $cn - d$.

Avoiding pitfalls

It is easy to err in the use of asymptotic notation. For example, in the recurrence (4.19) we can falsely "prove" $T(n) = O(n)$ by guessing $T(n) \leq cn$ and then arguing

$$\begin{aligned}
T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\
&\leq cn + n \\
&= O(n) , \qquad \Longleftarrow \textit{wrong!!}
\end{aligned}$$

since c is a constant. The error is that we have not proved the *exact form* of the inductive hypothesis, that is, that $T(n) \leq cn$. We therefore will explicitly prove that $T(n) \leq cn$ when we want to show that $T(n) = O(n)$.

Changing variables

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. As an example, consider the recurrence

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n ,$$

which looks difficult. We can simplify this recurrence, though, with a change of variables. For convenience, we shall not worry about rounding off values, such as $\sqrt{n}$, to be integers. Renaming $m = \lg n$ yields

$$T(2^m) = 2T(2^{m/2}) + m .$$

We can now rename $S(m) = T(2^m)$ to produce the new recurrence

$$S(m) = 2S(m/2) + m ,$$

which is very much like recurrence (4.19). Indeed, this new recurrence has the same solution: $S(m) = O(m \lg m)$. Changing back from $S(m)$ to $T(n)$, we obtain

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n) .$$

Exercises

4.3-1
Show that the solution of $T(n) = T(n-1) + n$ is $O(n^2)$.

4.3-2
Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$.

4.3-3
We saw that the solution of $T(n) = 2T(\lfloor n/2 \rfloor) + n$ is $O(n \lg n)$. Show that the solution of this recurrence is also $\Omega(n \lg n)$. Conclude that the solution is $\Theta(n \lg n)$.

4.3-4
Show that by making a different inductive hypothesis, we can overcome the difficulty with the boundary condition $T(1) = 1$ for recurrence (4.19) without adjusting the boundary conditions for the inductive proof.

4.3-5
Show that $\Theta(n \lg n)$ is the solution to the "exact" recurrence (4.3) for merge sort.

4.3-6
Show that the solution to $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$.

4.3-7
Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/3) + n$ is $T(n) = \Theta(n^{\log_3 4})$. Show that a substitution proof with the assumption $T(n) \le cn^{\log_3 4}$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

4.3-8
Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/2) + n^2$ is $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \le cn^2$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

4.3-9

Solve the recurrence $T(n) = 3T(\sqrt{n}) + \lg n$ by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integral.

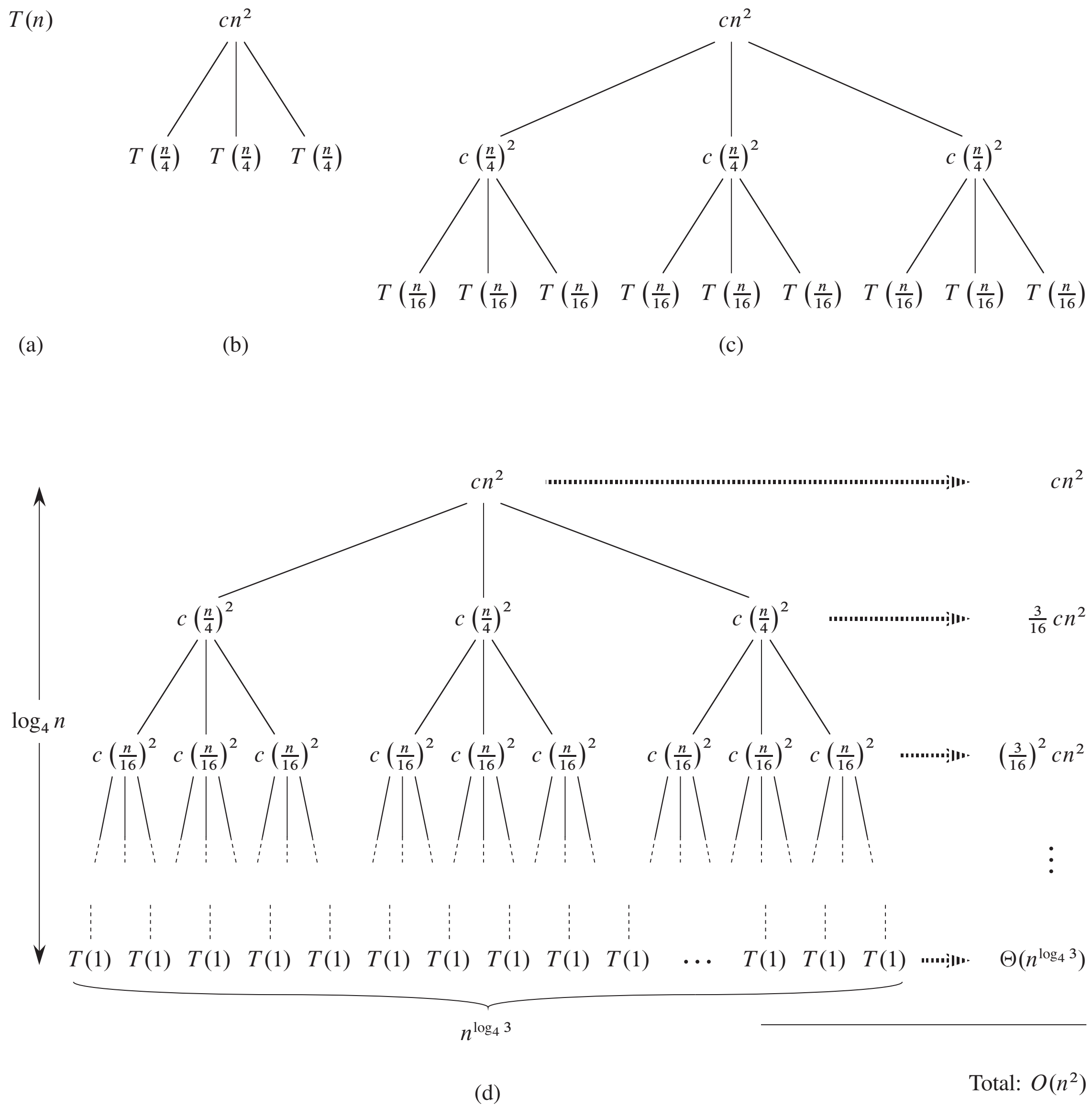
4.4 The recursion-tree method for solving recurrences

Although you can use the substitution method to provide a succinct proof that a solution to a recurrence is correct, you might have trouble coming up with a good guess. Drawing out a recursion tree, as we did in our analysis of the merge sort recurrence in Section 2.3.2, serves as a straightforward way to devise a good guess. In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

A recursion tree is best used to generate a good guess, which you can then verify by the substitution method. When using a recursion tree to generate a good guess, you can often tolerate a small amount of "sloppiness," since you will be verifying your guess later on. If you are very careful when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence. In this section, we will use recursion trees to generate good guesses, and in Section 4.6, we will use recursion trees directly to prove the theorem that forms the basis of the master method.

For example, let us see how a recursion tree would provide a good guess for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We start by focusing on finding an upper bound for the solution. Because we know that floors and ceilings usually do not matter when solving recurrences (here's an example of sloppiness that we can tolerate), we create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$, having written out the implied constant coefficient $c > 0$.

Figure 4.5 shows how we derive the recursion tree for $T(n) = 3T(n/4) + cn^2$. For convenience, we assume that n is an exact power of 4 (another example of tolerable sloppiness) so that all subproblem sizes are integers. Part (a) of the figure shows $T(n)$, which we expand in part (b) into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size $n/4$. Part (c) shows this process carried one step further by expanding each node with cost $T(n/4)$ from part (b). The cost for each of the three children of the root is $c(n/4)^2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.



$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$T(1)$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
Total: $O(n^2)$
(d)

Figure 4.5 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition. How far from the root do we reach one? The subproblem size for a node at depth i is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$).

Next we determine the cost at each level of the tree. Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3}T(1)$, which is $\Theta(n^{\log_4 3})$, since we assume that $T(1)$ is a constant.

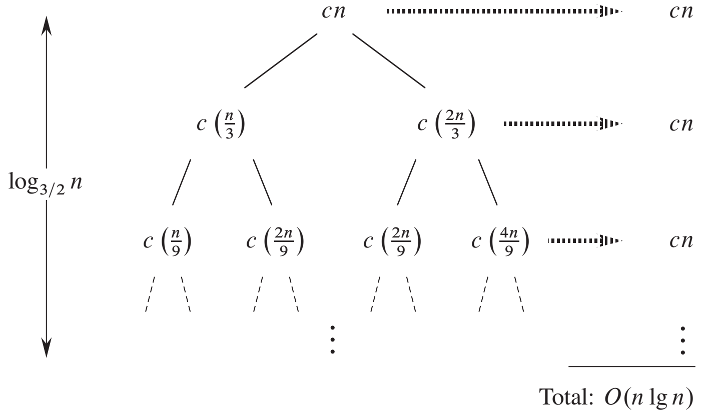
Now we add up the costs over all levels to determine the cost for the entire tree:

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(by equation (A.5))} .
\end{aligned}$$

This last formula looks somewhat messy until we realize that we can again take advantage of small amounts of sloppiness and use an infinite decreasing geometric series as an upper bound. Backing up one step and applying equation (A.6), we have

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}$$

Thus, we have derived a guess of $T(n) = O(n^2)$ for our original recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. In this example, the coefficients of cn^2 form a decreasing geometric series and, by equation (A.6), the sum of these coefficients



cn
cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
cn
$\log_{3/2} n$
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
Total: $O(n \lg n)$

Figure 4.6 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

is bounded from above by the constant 16/13. Since the root's contribution to the total cost is cn^2, the root contributes a constant fraction of the total cost. In other words, the cost of the root dominates the total cost of the tree.

In fact, if $O(n^2)$ is indeed an upper bound for the recurrence (as we shall verify in a moment), then it must be a tight bound. Why? The first recursive call contributes a cost of $\Theta(n^2)$, and so $\Omega(n^2)$ must be a lower bound for the recurrence.

Now we can use the substitution method to verify that our guess was correct, that is, $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We want to show that $T(n) \le dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\,\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2 , \end{aligned}$$

where the last step holds as long as $d \ge (16/13)c$.

In another, more intricate, example, Figure 4.6 shows the recursion tree for

$$T(n) = T(n/3) + T(2n/3) + O(n) .$$

(Again, we omit floor and ceiling functions for simplicity.) As before, we let c represent the constant factor in the $O(n)$ term. When we add the values across the levels of the recursion tree shown in the figure, we get a value of cn for every level.

The longest simple path from the root to a leaf is $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level, or $O(cn \log_{3/2} n) = O(n \lg n)$. Figure 4.6 shows only the top levels of the recursion tree, however, and not every level in the tree contributes a cost of cn. Consider the cost of the leaves. If this recursion tree were a complete binary tree of height $\log_{3/2} n$, there would be $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ leaves. Since the cost of each leaf is a constant, the total cost of all leaves would then be $\Theta(n^{\log_{3/2} 2})$ which, since $\log_{3/2} 2$ is a constant strictly greater than 1, is $\omega(n \lg n)$. This recursion tree is not a complete binary tree, however, and so it has fewer than $n^{\log_{3/2} 2}$ leaves. Moreover, as we go down from the root, more and more internal nodes are absent. Consequently, levels toward the bottom of the recursion tree contribute less than cn to the total cost. We could work out an accurate accounting of all costs, but remember that we are just trying to come up with a guess to use in the substitution method. Let us tolerate the sloppiness and attempt to show that a guess of $O(n \lg n)$ for the upper bound is correct.

Indeed, we can use the substitution method to verify that $O(n \lg n)$ is an upper bound for the solution to the recurrence. We show that $T(n) \le dn \lg n$, where d is a suitable positive constant. We have

$$\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n \ ,
\end{aligned}$$

as long as $d \ge c/(\lg 3 - (2/3))$. Thus, we did not need to perform a more accurate accounting of costs in the recursion tree.

Exercises

4.4-1

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Use the substitution method to verify your answer.

4.4-2

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$. Use the substitution method to verify your answer.

4.4-3
Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 4T(n/2 + 2) + n$. Use the substitution method to verify your answer.

4.4-4
Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 2T(n - 1) + 1$. Use the substitution method to verify your answer.

4.4-5
Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n-1)+T(n/2)+n$. Use the substitution method to verify your answer.

4.4-6
Argue that the solution to the recurrence $T(n) = T(n/3)+T(2n/3)+cn$, where c is a constant, is $\Omega(n \lg n)$ by appealing to a recursion tree.

4.4-7
Draw the recursion tree for $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, where c is a constant, and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.

4.4-8
Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(n - a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

4.4-9
Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

4.5 The master method for solving recurrences

The master method provides a "cookbook" method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n) , \tag{4.20}$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. To use the master method, you will need to memorize three cases, but then you will be able to solve many recurrences quite easily, often without pencil and paper.

The recurrence (4.20) describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The function $f(n)$ encompasses the cost of dividing the problem and combining the results of the subproblems. For example, the recurrence arising from Strassen's algorithm has $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$.

As a matter of technical correctness, the recurrence is not actually well defined, because n/b might not be an integer. Replacing each of the a terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ will not affect the asymptotic behavior of the recurrence, however. (We will prove this assertion in the next section.) We normally find it convenient, therefore, to omit the floor and ceiling functions when writing divide-and-conquer recurrences of this form.

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Beyond this intuition, you need to be aware of some technicalities. In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially* smaller.

That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant $\epsilon > 0$. In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that $af(n/b) \le cf(n)$. This condition is satisfied by most of the polynomially bounded functions that we shall encounter.

Note that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you cannot use the master method to solve the recurrence.

Using the master method

To use the master method, we simply determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider

$$T(n) = 9T(n/3) + n \; .$$

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

$$T(n) = T(2n/3) + 1 ,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence

$$T(n) = 3T(n/4) + n \lg n \; ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

The master method does not apply to the recurrence

$$T(n) = 2T(n/2) + n \lg n \; ,$$

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since

$f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.6-2 for a solution.)

Let's use the master method to solve the recurrences we saw in Sections 4.1 and 4.2. Recurrence (4.7),

$$T(n) = 2T(n/2) + \Theta(n) ,$$

characterizes the running times of the divide-and-conquer algorithm for both the maximum-subarray problem and merge sort. (As is our practice, we omit stating the base case in the recurrence.) Here, we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$, and thus we have that $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies, since $f(n) = \Theta(n)$, and so we have the solution $T(n) = \Theta(n \lg n)$.

Recurrence (4.17),

$$T(n) = 8T(n/2) + \Theta(n^2) ,$$

describes the running time of the first divide-and-conquer algorithm that we saw for matrix multiplication. Now we have $a = 8$, $b = 2$, and $f(n) = \Theta(n^2)$, and so $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than $f(n)$ (that is, $f(n) = O(n^{3-\epsilon})$ for $\epsilon = 1$), case 1 applies, and $T(n) = \Theta(n^3)$.

Finally, consider recurrence (4.18),

$$T(n) = 7T(n/2) + \Theta(n^2) ,$$

which describes the running time of Strassen's algorithm. Here, we have $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$, and thus $n^{\log_b a} = n^{\log_2 7}$. Rewriting $\log_2 7$ as $\lg 7$ and recalling that $2.80 < \lg 7 < 2.81$, we see that $f(n) = O(n^{\lg 7 - \epsilon})$ for $\epsilon = 0.8$. Again, case 1 applies, and we have the solution $T(n) = \Theta(n^{\lg 7})$.

Exercises

4.5-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1$.

b. $T(n) = 2T(n/4) + \sqrt{n}$.

c. $T(n) = 2T(n/4) + n$.

d. $T(n) = 2T(n/4) + n^2$.

4.5-2

Professor Caesar wishes to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into pieces of size $n/4 \times n/4$, and the divide and combine steps together will take $\Theta(n^2)$ time. He needs to determine how many subproblems his algorithm has to create in order to beat Strassen's algorithm. If his algorithm creates a subproblems, then the recurrence for the running time $T(n)$ becomes $T(n) = aT(n/4) + \Theta(n^2)$. What is the largest integer value of a for which Professor Caesar's algorithm would be asymptotically faster than Strassen's algorithm?

4.5-3

Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$. (See Exercise 2.3-5 for a description of binary search.)

4.5-4

Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \lg n$? Why or why not? Give an asymptotic upper bound for this recurrence.

4.5-5 ⋆

Consider the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of constants $a \geq 1$ and $b > 1$ and a function $f(n)$ that satisfies all the conditions in case 3 of the master theorem except the regularity condition.

⋆ 4.6 Proof of the master theorem

This section contains a proof of the master theorem (Theorem 4.1). You do not need to understand the proof in order to apply the master theorem.

The proof appears in two parts. The first part analyzes the master recurrence (4.20), under the simplifying assumption that $T(n)$ is defined only on exact powers of $b > 1$, that is, for $n = 1, b, b^2, \ldots$. This part gives all the intuition needed to understand why the master theorem is true. The second part shows how to extend the analysis to all positive integers n; it applies mathematical technique to the problem of handling floors and ceilings.

In this section, we shall sometimes abuse our asymptotic notation slightly by using it to describe the behavior of functions that are defined only over exact powers of b. Recall that the definitions of asymptotic notations require that bounds be proved for all sufficiently large numbers, not just those that are powers of b. Since we could make new asymptotic notations that apply only to the set $\{b^i : i = 0, 1, 2, \ldots\}$, instead of to the nonnegative numbers, this abuse is minor.

Nevertheless, we must always be on guard when we use asymptotic notation over a limited domain lest we draw improper conclusions. For example, proving that $T(n) = O(n)$ when n is an exact power of 2 does not guarantee that $T(n) = O(n)$. The function $T(n)$ could be defined as

$$T(n) = \begin{cases} n & \text{if } n = 1, 2, 4, 8, \ldots , \\ n^2 & \text{otherwise} , \end{cases}$$

in which case the best upper bound that applies to all values of n is $T(n) = O(n^2)$. Because of this sort of drastic consequence, we shall never use asymptotic notation over a limited domain without making it absolutely clear from the context that we are doing so.

4.6.1 The proof for exact powers

The first part of the proof of the master theorem analyzes the recurrence (4.20)

$$T(n) = aT(n/b) + f(n) ,$$

for the master method, under the assumption that n is an exact power of $b > 1$, where b need not be an integer. We break the analysis into three lemmas. The first reduces the problem of solving the master recurrence to the problem of evaluating an expression that contains a summation. The second determines bounds on this summation. The third lemma puts the first two together to prove a version of the master theorem for the case in which n is an exact power of b.

Lemma 4.2
Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ aT(n/b) + f(n) & \text{if } n = b^i , \end{cases}$$

where i is a positive integer. Then

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) . \tag{4.21}$$

Proof We use the recursion tree in Figure 4.7. The root of the tree has cost $f(n)$, and it has a children, each with cost $f(n/b)$. (It is convenient to think of a as being

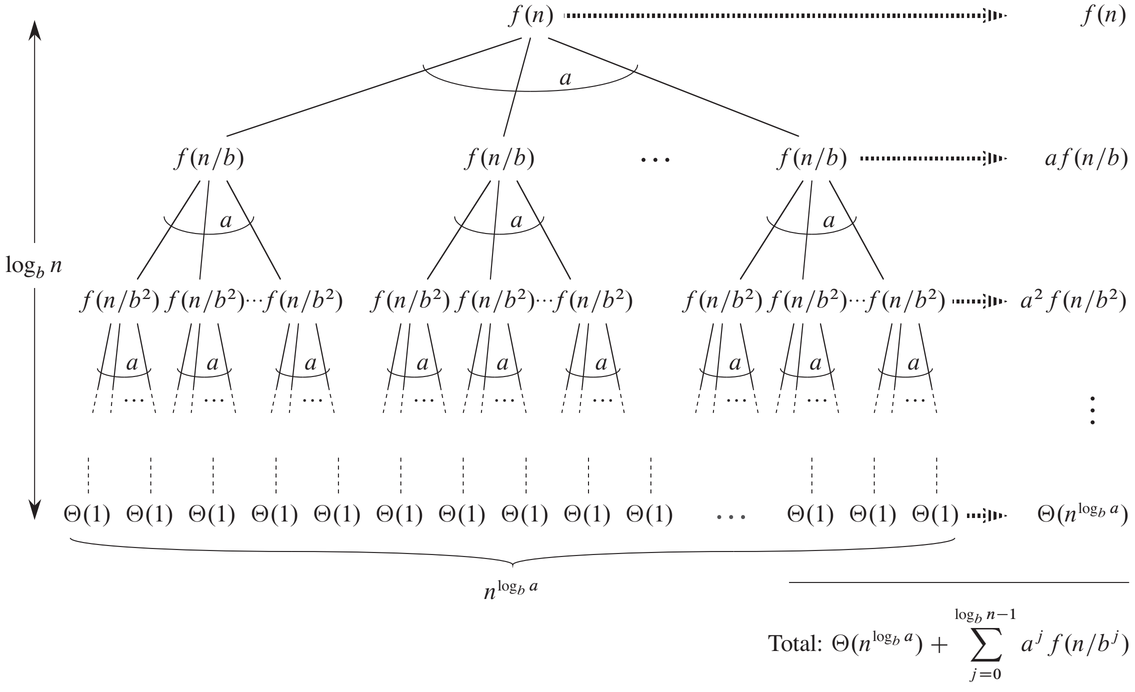


Figure 4.7 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of the nodes at each depth is shown at the right, and their sum is given in equation (4.21).

an integer, especially when visualizing the recursion tree, but the mathematics does not require it.) Each of these children has a children, making a^2 nodes at depth 2, and each of the a children has cost $f(n/b^2)$. In general, there are a^j nodes at depth j, and each has cost $f(n/b^j)$. The cost of each leaf is $T(1) = \Theta(1)$, and each leaf is at depth $\log_b n$, since $n/b^{\log_b n} = 1$. There are $a^{\log_b n} = n^{\log_b a}$ leaves in the tree.

We can obtain equation (4.21) by summing the costs of the nodes at each depth in the tree, as shown in the figure. The cost for all internal nodes at depth j is $a^j f(n/b^j)$, and so the total cost of all internal nodes is

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) .$$

In the underlying divide-and-conquer algorithm, this sum represents the costs of dividing problems into subproblems and then recombining the subproblems. The cost of all the leaves, which is the cost of doing all $n^{\log_b a}$ subproblems of size 1, is $\Theta(n^{\log_b a})$. ■

In terms of the recursion tree, the three cases of the master theorem correspond to cases in which the total cost of the tree is (1) dominated by the costs in the leaves, (2) evenly distributed among the levels of the tree, or (3) dominated by the cost of the root.

The summation in equation (4.21) describes the cost of the dividing and combining steps in the underlying divide-and-conquer algorithm. The next lemma provides asymptotic bounds on the summation's growth.

Lemma 4.3

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{4.22}$$

has the following asymptotic bounds for exact powers of b:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $g(n) = O(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $af(n/b) \leq cf(n)$ for some constant $c < 1$ and for all sufficiently large n, then $g(n) = \Theta(f(n))$.

Proof For case 1, we have $f(n) = O(n^{\log_b a - \epsilon})$, which implies that $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$. Substituting into equation (4.22) yields

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) . \tag{4.23}$$

We bound the summation within the O-notation by factoring out terms and simplifying, which leaves an increasing geometric series:

$$\begin{aligned}
\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon} &= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j \\
&= n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right)
\end{aligned}$$

$$= n^{\log_b a-\epsilon}\left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right) .$$

Since b and ϵ are constants, we can rewrite the last expression as $n^{\log_b a-\epsilon}O(n^\epsilon) = O(n^{\log_b a})$. Substituting this expression for the summation in equation (4.23) yields

$$g(n) = O(n^{\log_b a}) ,$$

thereby proving case 1.

Because case 2 assumes that $f(n) = \Theta(n^{\log_b a})$, we have that $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Substituting into equation (4.22) yields

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) . \tag{4.24}$$

We bound the summation within the Θ-notation as in case 1, but this time we do not obtain a geometric series. Instead, we discover that every term of the summation is the same:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a}\sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a}\log_b n .\end{aligned}$$

Substituting this expression for the summation in equation (4.24) yields

$$\begin{aligned}g(n) &= \Theta(n^{\log_b a}\log_b n) \\ &= \Theta(n^{\log_b a}\lg n) ,\end{aligned}$$

proving case 2.

We prove case 3 similarly. Since $f(n)$ appears in the definition (4.22) of $g(n)$ and all terms of $g(n)$ are nonnegative, we can conclude that $g(n) = \Omega(f(n))$ for exact powers of b. We assume in the statement of the lemma that $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n. We rewrite this assumption as $f(n/b) \le (c/a)f(n)$ and iterate j times, yielding $f(n/b^j) \le (c/a)^j f(n)$ or, equivalently, $a^j f(n/b^j) \le c^j f(n)$, where we assume that the values we iterate on are sufficiently large. Since the last, and smallest, such value is n/b^{j-1}, it is enough to assume that n/b^{j-1} is sufficiently large.

Substituting into equation (4.22) and simplifying yields a geometric series, but unlike the series in case 1, this one has decreasing terms. We use an $O(1)$ term to capture the terms that are not covered by our assumption that n is sufficiently large:

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\
&\le \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1) \\
&\le f(n)\sum_{j=0}^{\infty} c^j + O(1) \\
&= f(n)\left(\frac{1}{1-c}\right) + O(1) \\
&= O(f(n)) ,
\end{aligned}$$

since c is a constant. Thus, we can conclude that $g(n) = \Theta(f(n))$ for exact powers of b. With case 3 proved, the proof of the lemma is complete. ■

We can now prove a version of the master theorem for the case in which n is an exact power of b.

Lemma 4.4
Let $a \ge 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ aT(n/b) + f(n) & \text{if } n = b^i , \end{cases}$$

where i is a positive integer. Then $T(n)$ has the following asymptotic bounds for exact powers of b:

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a}\lg n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Proof We use the bounds in Lemma 4.3 to evaluate the summation (4.21) from Lemma 4.2. For case 1, we have

$$\begin{aligned}
T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\
&= \Theta(n^{\log_b a}) ,
\end{aligned}$$

and for case 2,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n) . \end{aligned}$$

For case 3,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)) , \end{aligned}$$

because $f(n) = \Omega(n^{\log_b a+\epsilon})$. ■

4.6.2 Floors and ceilings

To complete the proof of the master theorem, we must now extend our analysis to the situation in which floors and ceilings appear in the master recurrence, so that the recurrence is defined for all integers, not for just exact powers of b. Obtaining a lower bound on

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.25}$$

and an upper bound on

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.26}$$

is routine, since we can push through the bound $\lceil n/b \rceil \geq n/b$ in the first case to yield the desired result, and we can push through the bound $\lfloor n/b \rfloor \leq n/b$ in the second case. We use much the same technique to lower-bound the recurrence (4.26) as to upper-bound the recurrence (4.25), and so we shall present only this latter bound.

We modify the recursion tree of Figure 4.7 to produce the recursion tree in Figure 4.8. As we go down in the recursion tree, we obtain a sequence of recursive invocations on the arguments

$$\begin{aligned} &n , \\ &\lceil n/b \rceil , \\ &\lceil \lceil n/b \rceil /b \rceil , \\ &\lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil , \\ &\qquad \vdots \end{aligned}$$

Let us denote the jth element in the sequence by n_j, where

$$n_j = \begin{cases} n & \text{if } j = 0 , \\ \lceil n_{j-1}/b \rceil & \text{if } j > 0 . \end{cases} \tag{4.27}$$

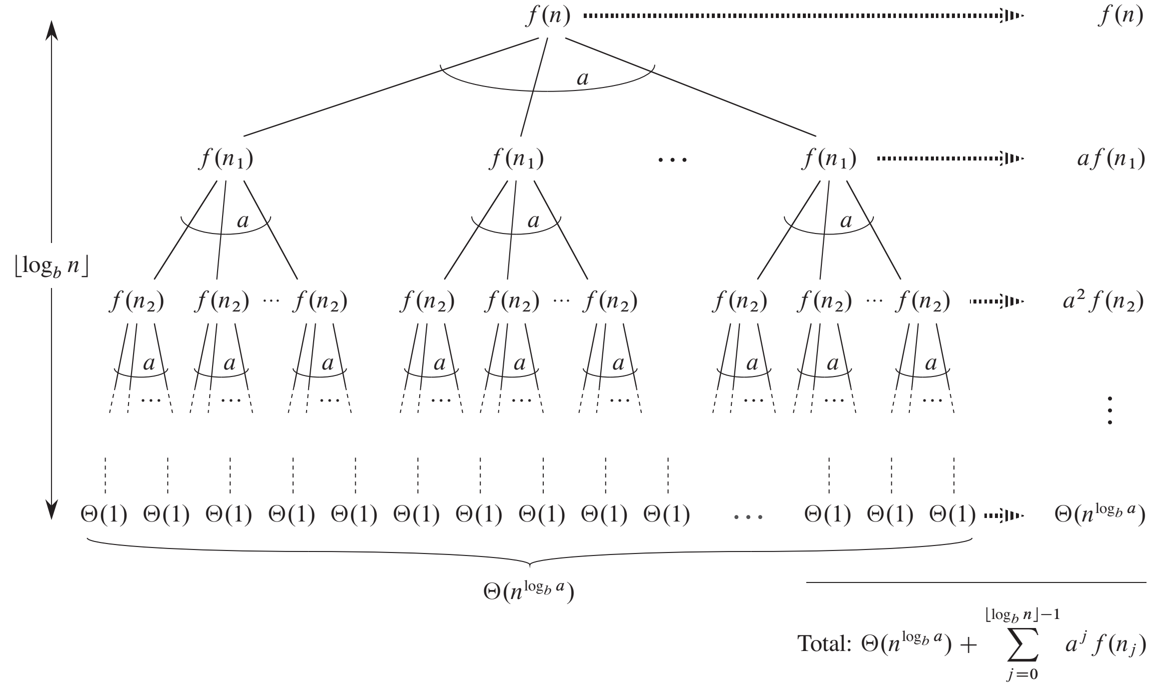

$f(n)$
a
$f(n_1)$
$a f(n_1)$
$f(n_2)$
$a^2 f(n_2)$
$\lfloor \log_b n \rfloor$
$\Theta(1)$
$\Theta(n^{\log_b a})$
Total: $\Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j)$

Figure 4.8 The recursion tree generated by $T(n) = aT(\lceil n/b \rceil) + f(n)$. The recursive argument n_j is given by equation (4.27).

Our first goal is to determine the depth k such that n_k is a constant. Using the inequality $\lceil x \rceil \le x + 1$, we obtain

$$\begin{aligned} n_0 &\le n , \\ n_1 &\le \frac{n}{b} + 1 , \\ n_2 &\le \frac{n}{b^2} + \frac{1}{b} + 1 , \\ n_3 &\le \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1 , \\ &\vdots \end{aligned}$$

In general, we have

$$\begin{aligned} n_j &\le \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\ &< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\ &= \frac{n}{b^j} + \frac{b}{b-1} \, . \end{aligned}$$

Letting $j = \lfloor \log_b n \rfloor$, we obtain

$$\begin{aligned} n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \\ &< \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\ &= \frac{n}{n/b} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} \\ &= O(1) \, , \end{aligned}$$

and thus we see that at depth $\lfloor \log_b n \rfloor$, the problem size is at most a constant.

From Figure 4.8, we see that

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \, , \tag{4.28}$$

which is much the same as equation (4.21), except that n is an arbitrary integer and not restricted to be an exact power of b.

We can now evaluate the summation

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.29}$$

from equation (4.28) in a manner analogous to the proof of Lemma 4.3. Beginning with case 3, if $af(\lceil n/b \rceil) \le cf(n)$ for $n > b + b/(b-1)$, where $c < 1$ is a constant, then it follows that $a^j f(n_j) \le c^j f(n)$. Therefore, we can evaluate the sum in equation (4.29) just as in Lemma 4.3. For case 2, we have $f(n) = \Theta(n^{\log_b a})$. If we can show that $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, then the proof for case 2 of Lemma 4.3 will go through. Observe that $j \le \lfloor \log_b n \rfloor$ implies $b^j/n \le 1$. The bound $f(n) = O(n^{\log_b a})$ implies that there exists a constant $c > 0$ such that for all sufficiently large n_j,

$$\begin{aligned}
f(n_j) &\le c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\
&= c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a} \\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a} \\
&\le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \\
&= O\left(\frac{n^{\log_b a}}{a^j}\right) ,
\end{aligned}$$

since $c(1 + b/(b-1))^{\log_b a}$ is a constant. Thus, we have proved case 2. The proof of case 1 is almost identical. The key is to prove the bound $f(n_j) = O(n^{\log_b a - \epsilon})$, which is similar to the corresponding proof of case 2, though the algebra is more intricate.

We have now proved the upper bounds in the master theorem for all integers n. The proof of the lower bounds is similar.

Exercises

4.6-1 ★

Give a simple and exact expression for n_j in equation (4.27) for the case in which b is a positive integer instead of an arbitrary real number.

4.6-2 ★

Show that if $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \ge 0$, then the master recurrence has solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. For simplicity, confine your analysis to exact powers of b.

4.6-3 ★

Show that case 3 of the master theorem is overstated, in the sense that the regularity condition $af(n/b) \le cf(n)$ for some constant $c < 1$ implies that there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Problems

4-1 Recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \le 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(n/2) + n^4$.

b. $T(n) = T(7n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-2) + n^2$.

4-2 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time $= \Theta(1)$.
2. An array is passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(q - p + 1)$ if the subarray $A[p \,..\, q]$ is passed.

a. Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.

b. Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.

4-3 More recurrence examples
Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small n. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 4T(n/3) + n \lg n$.

b. $T(n) = 3T(n/3) + n/\lg n$.

c. $T(n) = 4T(n/2) + n^2\sqrt{n}$.

d. $T(n) = 3T(n/3 - 2) + n/2$.

e. $T(n) = 2T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n-1) + 1/n$.

h. $T(n) = T(n-1) + \lg n$.

i. $T(n) = T(n-2) + 1/\lg n$.

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-4 Fibonacci numbers
This problem develops properties of the Fibonacci numbers, which are defined by recurrence (3.22). We shall use the technique of generating functions to solve the Fibonacci recurrence. Define the ***generating function*** (or ***formal power series***) $\mathcal{F}$ as

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots ,\end{aligned}$$

where F_i is the ith Fibonacci number.

a. Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Show that

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1-z-z^2} \\ &= \frac{z}{(1-\phi z)(1-\hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right) ,\end{aligned}$$

where

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$$

and

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\ldots .$$

c. Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) z^i .$$

d. Use part (c) to prove that $F_i = \phi^i/\sqrt{5}$ for $i > 0$, rounded to the nearest integer. (*Hint:* Observe that $|\hat{\phi}| < 1$.)

4-5 Chip testing

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

a. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

b. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

c. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

4-6 Monge arrays

An $m \times n$ array A of real numbers is a ***Monge array*** if for all i, j, k, and l such that $1 \le i < k \le m$ and $1 \le j < l \le n$, we have

$$A[i,j] + A[k,l] \le A[i,l] + A[k,j] .$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and the columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove that an array is Monge if and only if for all $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots, n-1$, we have

$$A[i,j] + A[i+1,j+1] \le A[i,j+1] + A[i+1,j] .$$

(*Hint:* For the "if" part, use induction separately on rows and columns.)

b. The following array is not Monge. Change one element in order to make it Monge. (*Hint:* Use part (a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \le f(2) \le \cdots \le f(m)$ for any $m \times n$ Monge array.

d. Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A:

> Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in $O(m + n)$ time.

e. Write the recurrence describing the running time of the algorithm described in part (d). Show that its solution is $O(m + n \log m)$.

Chapter notes

Divide-and-conquer as a technique for designing algorithms dates back to at least 1962 in an article by Karatsuba and Ofman [194]. It might have been used well before then, however; according to Heideman, Johnson, and Burrus [163], C. F. Gauss devised the first fast Fourier transform algorithm in 1805, and Gauss's formulation breaks the problem into smaller subproblems whose solutions are combined.

The maximum-subarray problem in Section 4.1 is a minor variation on a problem studied by Bentley [43, Chapter 7].

Strassen's algorithm [325] caused much excitement when it was published in 1969. Before then, few imagined the possibility of an algorithm asymptotically faster than the basic SQUARE-MATRIX-MULTIPLY procedure. The asymptotic upper bound for matrix multiplication has been improved since then. The most asymptotically efficient algorithm for multiplying $n \times n$ matrices to date, due to Coppersmith and Winograd [78], has a running time of $O(n^{2.376})$. The best lower bound known is just the obvious $\Omega(n^2)$ bound (obvious because we must fill in n^2 elements of the product matrix).

From a practical point of view, Strassen's algorithm is often not the method of choice for matrix multiplication, for four reasons:

1. The constant factor hidden in the $\Theta(n^{\lg 7})$ running time of Strassen's algorithm is larger than the constant factor in the $\Theta(n^3)$-time SQUARE-MATRIX-MULTIPLY procedure.
2. When the matrices are sparse, methods tailored for sparse matrices are faster.

3. Strassen's algorithm is not quite as numerically stable as SQUARE-MATRIX-MULTIPLY. In other words, because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in SQUARE-MATRIX-MULTIPLY.

4. The submatrices formed at the levels of recursion consume space.

The latter two reasons were mitigated around 1990. Higham [167] demonstrated that the difference in numerical stability had been overemphasized; although Strassen's algorithm is too numerically unstable for some applications, it is within acceptable limits for others. Bailey, Lee, and Simon [32] discuss techniques for reducing the memory requirements for Strassen's algorithm.

In practice, fast matrix-multiplication implementations for dense matrices use Strassen's algorithm for matrix sizes above a "crossover point," and they switch to a simpler method once the subproblem size reduces to below the crossover point. The exact value of the crossover point is highly system dependent. Analyses that count operations but ignore effects from caches and pipelining have produced crossover points as low as $n = 8$ (by Higham [167]) or $n = 12$ (by Huss-Lederman et al. [186]). D'Alberto and Nicolau [81] developed an adaptive scheme, which determines the crossover point by benchmarking when their software package is installed. They found crossover points on various systems ranging from $n = 400$ to $n = 2150$, and they could not find a crossover point on a couple of systems.

Recurrences were studied as early as 1202 by L. Fibonacci, for whom the Fibonacci numbers are named. A. De Moivre introduced the method of generating functions (see Problem 4-4) for solving recurrences. The master method is adapted from Bentley, Haken, and Saxe [44], which provides the extended method justified by Exercise 4.6-2. Knuth [209] and Liu [237] show how to solve linear recurrences using the method of generating functions. Purdom and Brown [287] and Graham, Knuth, and Patashnik [152] contain extended discussions of recurrence solving.

Several researchers, including Akra and Bazzi [13], Roura [299], Verma [346], and Yap [360], have given methods for solving more general divide-and-conquer recurrences than are solved by the master method. We describe the result of Akra and Bazzi here, as modified by Leighton [228]. The Akra-Bazzi method works for recurrences of the form

$$T(x) = \begin{cases} \Theta(1) & \text{if } 1 \le x \le x_0 , \\ \sum_{i=1}^{k} a_i T(b_i x) + f(x) & \text{if } x > x_0 , \end{cases} \tag{4.30}$$

where

- $x \ge 1$ is a real number,
- x_0 is a constant such that $x_0 \ge 1/b_i$ and $x_0 \ge 1/(1 - b_i)$ for $i = 1, 2, \ldots, k$,
- a_i is a positive constant for $i = 1, 2, \ldots, k$,

- b_i is a constant in the range $0 < b_i < 1$ for $i = 1, 2, \ldots, k$,
- $k \geq 1$ is an integer constant, and
- $f(x)$ is a nonnegative function that satisfies the ***polynomial-growth condition***: there exist positive constants c_1 and c_2 such that for all $x \geq 1$, for $i = 1, 2, \ldots, k$, and for all u such that $b_i x \leq u \leq x$, we have $c_1 f(x) \leq f(u) \leq c_2 f(x)$. (If $|f'(x)|$ is upper-bounded by some polynomial in x, then $f(x)$ satisfies the polynomial-growth condition. For example, $f(x) = x^\alpha \lg^\beta x$ satisfies this condition for any real constants α and β.)

Although the master method does not apply to a recurrence such as $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, the Akra-Bazzi method does. To solve the recurrence (4.30), we first find the unique real number p such that $\sum_{i=1}^{k} a_i b_i^p = 1$. (Such a p always exists.) The solution to the recurrence is then

$$T(n) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)}{u^{p+1}}\, du\right)\right) .$$

The Akra-Bazzi method can be somewhat difficult to use, but it serves in solving recurrences that model division of the problem into substantially unequally sized subproblems. The master method is simpler to use, but it applies only when subproblem sizes are equal.

Randomized Algorithms

5 Probabilistic Analysis and Randomized Algorithms

This chapter introduces probabilistic analysis and randomized algorithms. If you are unfamiliar with the basics of probability theory, you should read Appendix C, which reviews this material. We shall revisit probabilistic analysis and randomized algorithms several times throughout this book.

5.1 The hiring problem

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency sends you one candidate each day. You interview that person and then decide either to hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and pay a substantial hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be.

The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring in pseudocode. It assumes that the candidates for the office assistant job are numbered 1 through n. The procedure assumes that you are able to, after interviewing candidate i, determine whether candidate i is the best candidate you have seen so far. To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

```
HIRE-ASSISTANT(n)
1  best = 0            // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          hire candidate i
```

The cost model for this problem differs from the model described in Chapter 2. We focus not on the running time of HIRE-ASSISTANT, but instead on the costs incurred by interviewing and hiring. On the surface, analyzing the cost of this algorithm may seem very different from analyzing the running time of, say, merge sort. The analytical techniques used, however, are identical whether we are analyzing cost or running time. In either case, we are counting the number of times certain basic operations are executed.

Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h. Letting m be the number of people hired, the total cost associated with this algorithm is $O(c_i n + c_h m)$. No matter how many people we hire, we always interview n candidates and thus always incur the cost $c_i n$ associated with interviewing. We therefore concentrate on analyzing $c_h m$, the hiring cost. This quantity varies with each run of the algorithm.

This scenario serves as a model for a common computational paradigm. We often need to find the maximum or minimum value in a sequence by examining each element of the sequence and maintaining a current "winner." The hiring problem models how often we update our notion of which element is currently winning.

Worst-case analysis

In the worst case, we actually hire every candidate that we interview. This situation occurs if the candidates come in strictly increasing order of quality, in which case we hire n times, for a total hiring cost of $O(c_h n)$.

Of course, the candidates do not always come in increasing order of quality. In fact, we have no idea about the order in which they arrive, nor do we have any control over this order. Therefore, it is natural to ask what we expect to happen in a typical or average case.

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm. Sometimes we use it to analyze other quantities, such as the hiring cost in procedure HIRE-ASSISTANT. In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs. Then we analyze our algorithm, computing an average-case running time, where we take the average over the distribution of the possible inputs. Thus we are, in effect, averaging the running time over all possible inputs. When reporting such a running time, we will refer to it as the ***average-case running time***.

We must be very careful in deciding on the distribution of inputs. For some problems, we may reasonably assume something about the set of all possible inputs, and then we can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, we cannot describe a reasonable input distribution, and in these cases we cannot use probabilistic analysis.

For the hiring problem, we can assume that the applicants come in a random order. What does that mean for this problem? We assume that we can compare any two candidates and decide which one is better qualified; that is, there is a total order on the candidates. (See Appendix B for the definition of a total order.) Thus, we can rank each candidate with a unique number from 1 through n, using $rank(i)$ to denote the rank of applicant i, and adopt the convention that a higher rank corresponds to a better qualified applicant. The ordered list $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ is a permutation of the list $\langle 1, 2, \ldots, n \rangle$. Saying that the applicants come in a random order is equivalent to saying that this list of ranks is equally likely to be any one of the $n!$ permutations of the numbers 1 through n. Alternatively, we say that the ranks form a ***uniform random permutation***; that is, each of the possible $n!$ permutations appears with equal probability.

Section 5.2 contains a probabilistic analysis of the hiring problem.

Randomized algorithms

In order to use probabilistic analysis, we need to know something about the distribution of the inputs. In many cases, we know very little about the input distribution. Even if we do know something about the distribution, we may not be able to model this knowledge computationally. Yet we often can use probability and randomness as a tool for algorithm design and analysis, by making the behavior of part of the algorithm random.

In the hiring problem, it may seem as if the candidates are being presented to us in a random order, but we have no way of knowing whether or not they really are. Thus, in order to develop a randomized algorithm for the hiring problem, we must have greater control over the order in which we interview the candidates. We will, therefore, change the model slightly. We say that the employment agency has n candidates, and they send us a list of the candidates in advance. On each day, we choose, randomly, which candidate to interview. Although we know nothing about the candidates (besides their names), we have made a significant change. Instead of relying on a guess that the candidates come to us in a random order, we have instead gained control of the process and enforced a random order.

More generally, we call an algorithm ***randomized*** if its behavior is determined not only by its input but also by values produced by a ***random-number generator***. We shall assume that we have at our disposal a random-number generator RANDOM. A call to RANDOM(a, b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM$(0, 1)$ produces 0 with probability $1/2$, and it produces 1 with probability $1/2$. A call to RANDOM$(3, 7)$ returns either 3, 4, 5, 6, or 7, each with probability $1/5$. Each integer returned by RANDOM is independent of the integers returned on previous calls. You may imagine RANDOM as rolling a $(b - a + 1)$-sided die to obtain its output. (In practice, most programming environments offer a ***pseudorandom-number generator***: a deterministic algorithm returning numbers that "look" statistically random.)

When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator. We distinguish these algorithms from those in which the input is random by referring to the running time of a randomized algorithm as an ***expected running time***. In general, we discuss the average-case running time when the probability distribution is over the inputs to the algorithm, and we discuss the expected running time when the algorithm itself makes random choices.

Exercises

5.1-1

Show that the assumption that we are always able to determine which candidate is best, in line 4 of procedure HIRE-ASSISTANT, implies that we know a total order on the ranks of the candidates.

5.1-2 ★

Describe an implementation of the procedure RANDOM(a, b) that only makes calls to RANDOM$(0, 1)$. What is the expected running time of your procedure, as a function of a and b?

5.1-3 ★

Suppose that you want to output 0 with probability $1/2$ and 1 with probability $1/2$. At your disposal is a procedure BIASED-RANDOM, that outputs either 0 or 1. It outputs 1 with some probability p and 0 with probability $1 - p$, where $0 < p < 1$, but you do not know what p is. Give an algorithm that uses BIASED-RANDOM as a subroutine, and returns an unbiased answer, returning 0 with probability $1/2$

and 1 with probability 1/2. What is the expected running time of your algorithm as a function of p?

5.2 Indicator random variables

In order to analyze many algorithms, including the hiring problem, we use indicator random variables. Indicator random variables provide a convenient method for converting between probabilities and expectations. Suppose we are given a sample space S and an event A. Then the ***indicator random variable*** $\mathrm{I}\{A\}$ associated with event A is defined as

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} , \\ 0 & \text{if } A \text{ does not occur} . \end{cases} \tag{5.1}$$

As a simple example, let us determine the expected number of heads that we obtain when flipping a fair coin. Our sample space is $S = \{H, T\}$, with $\Pr\{H\} = \Pr\{T\} = 1/2$. We can then define an indicator random variable X_H, associated with the coin coming up heads, which is the event H. This variable counts the number of heads obtained in this flip, and it is 1 if the coin comes up heads and 0 otherwise. We write

$$\begin{aligned} X_H &= \mathrm{I}\{H\} \\ &= \begin{cases} 1 & \text{if } H \text{ occurs} , \\ 0 & \text{if } T \text{ occurs} . \end{cases} \end{aligned}$$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable X_H:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 . \end{aligned}$$

Thus the expected number of heads obtained by one flip of a fair coin is 1/2. As the following lemma shows, the expected value of an indicator random variable associated with an event A is equal to the probability that A occurs.

Lemma 5.1
Given a sample space S and an event A in the sample space S, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof By the definition of an indicator random variable from equation (5.1) and the definition of expected value, we have

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\} , \end{aligned}$$

where $\overline{A}$ denotes $S - A$, the complement of A. ■

Although indicator random variables may seem cumbersome for an application such as counting the expected number of heads on a flip of a single coin, they are useful for analyzing situations in which we perform repeated random trials. For example, indicator random variables give us a simple way to arrive at the result of equation (C.37). In this equation, we compute the number of heads in n coin flips by considering separately the probability of obtaining 0 heads, 1 head, 2 heads, etc. The simpler method proposed in equation (C.38) instead uses indicator random variables implicitly. Making this argument more explicit, we let X_i be the indicator random variable associated with the event in which the ith flip comes up heads: $X_i = \mathrm{I}\{\text{the } i\text{th flip results in the event } H\}$. Let X be the random variable denoting the total number of heads in the n coin flips, so that

$$X = \sum_{i=1}^{n} X_i .$$

We wish to compute the expected number of heads, and so we take the expectation of both sides of the above equation to obtain

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] .$$

The above equation gives the expectation of the sum of n indicator random variables. By Lemma 5.1, we can easily compute the expectation of each of the random variables. By equation (C.21)—linearity of expectation—it is easy to compute the expectation of the sum: it equals the sum of the expectations of the n random variables. Linearity of expectation makes the use of indicator random variables a powerful analytical technique; it applies even when there is dependence among the random variables. We now can easily compute the expected number of heads:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 .
\end{aligned}$$

Thus, compared to the method used in equation (C.37), indicator random variables greatly simplify the calculation. We shall use indicator random variables throughout this book.

Analysis of the hiring problem using indicator random variables

Returning to the hiring problem, we now wish to compute the expected number of times that we hire a new office assistant. In order to use a probabilistic analysis, we assume that the candidates arrive in a random order, as discussed in the previous section. (We shall see in Section 5.3 how to remove this assumption.) Let X be the random variable whose value equals the number of times we hire a new office assistant. We could then apply the definition of expected value from equation (C.20) to obtain

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\} ,$$

but this calculation would be cumbersome. We shall instead use indicator random variables to greatly simplify the calculation.

To use indicator random variables, instead of computing $\mathrm{E}[X]$ by defining one variable associated with the number of times we hire a new office assistant, we define n variables related to whether or not each particular candidate is hired. In particular, we let X_i be the indicator random variable associated with the event in which the ith candidate is hired. Thus,

$$\begin{aligned}
X_i &= \mathrm{I}\{\text{candidate } i \text{ is hired}\} \\
&= \begin{cases} 1 & \text{if candidate } i \text{ is hired} , \\ 0 & \text{if candidate } i \text{ is not hired} , \end{cases}
\end{aligned}$$

and

$$X = X_1 + X_2 + \cdots + X_n . \tag{5.2}$$

By Lemma 5.1, we have that

$$\mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\} \ ,$$

and we must therefore compute the probability that lines 5–6 of HIRE-ASSISTANT are executed.

Candidate i is hired, in line 6, exactly when candidate i is better than each of candidates 1 through $i-1$. Because we have assumed that the candidates arrive in a random order, the first i candidates have appeared in a random order. Any one of these first i candidates is equally likely to be the best-qualified so far. Candidate i has a probability of $1/i$ of being better qualified than candidates 1 through $i-1$ and thus a probability of $1/i$ of being hired. By Lemma 5.1, we conclude that

$$\mathrm{E}[X_i] = 1/i \ . \tag{5.3}$$

Now we can compute $\mathrm{E}[X]$:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.2))} && (5.4)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} 1/i && \text{(by equation (5.3))} \\
&= \ln n + O(1) && \text{(by equation (A.7))} \ . && (5.5)
\end{aligned}$$

Even though we interview n people, we actually hire only approximately $\ln n$ of them, on average. We summarize this result in the following lemma.

Lemma 5.2
Assuming that the candidates are presented in a random order, algorithm HIRE-ASSISTANT has an average-case total hiring cost of $O(c_h \ln n)$.

Proof The bound follows immediately from our definition of the hiring cost and equation (5.5), which shows that the expected number of hires is approximately $\ln n$. ■

The average-case hiring cost is a significant improvement over the worst-case hiring cost of $O(c_h n)$.

Exercises

5.2-1
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly one time? What is the probability that you hire exactly n times?

5.2-2
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly twice?

5.2-3
Use indicator random variables to compute the expected value of the sum of n dice.

5.2-4
Use indicator random variables to solve the following problem, which is known as the ***hat-check problem***. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

5.2-5
Let $A[1\,..\,n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A. (See Problem 2-4 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

5.3 Randomized algorithms

In the previous section, we showed how knowing a distribution on the inputs can help us to analyze the average-case behavior of an algorithm. Many times, we do not have such knowledge, thus precluding an average-case analysis. As mentioned in Section 5.1, we may be able to use a randomized algorithm.

For a problem such as the hiring problem, in which it is helpful to assume that all permutations of the input are equally likely, a probabilistic analysis can guide the development of a randomized algorithm. Instead of assuming a distribution of inputs, we impose a distribution. In particular, before running the algorithm, we randomly permute the candidates in order to enforce the property that every permutation is equally likely. Although we have modified the algorithm, we still expect to hire a new office assistant approximately $\ln n$ times. But now we expect this to be the case for *any* input, rather than for inputs drawn from a particular distribution.

Let us further explore the distinction between probabilistic analysis and randomized algorithms. In Section 5.2, we claimed that, assuming that the candidates arrive in a random order, the expected number of times we hire a new office assistant is about $\ln n$. Note that the algorithm here is deterministic; for any particular input, the number of times a new office assistant is hired is always the same. Furthermore, the number of times we hire a new office assistant differs for different inputs, and it depends on the ranks of the various candidates. Since this number depends only on the ranks of the candidates, we can represent a particular input by listing, in order, the ranks of the candidates, i.e., $\langle rank(1), rank(2), \ldots, rank(n) \rangle$. Given the rank list $A_1 = \langle 1,2,3,4,5,6,7,8,9,10 \rangle$, a new office assistant is always hired 10 times, since each successive candidate is better than the previous one, and lines 5–6 are executed in each iteration. Given the list of ranks $A_2 = \langle 10,9,8,7,6,5,4,3,2,1 \rangle$, a new office assistant is hired only once, in the first iteration. Given a list of ranks $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6 \rangle$, a new office assistant is hired three times, upon interviewing the candidates with ranks 5, 8, and 10. Recalling that the cost of our algorithm depends on how many times we hire a new office assistant, we see that there are expensive inputs such as A_1, inexpensive inputs such as A_2, and moderately expensive inputs such as A_3.

Consider, on the other hand, the randomized algorithm that first permutes the candidates and then determines the best candidate. In this case, we randomize in the algorithm, not in the input distribution. Given a particular input, say A_3 above, we cannot say how many times the maximum is updated, because this quantity differs with each run of the algorithm. The first time we run the algorithm on A_3, it may produce the permutation A_1 and perform 10 updates; but the second time we run the algorithm, we may produce the permutation A_2 and perform only one update. The third time we run it, we may perform some other number of updates. Each time we run the algorithm, the execution depends on the random choices made and is likely to differ from the previous execution of the algorithm. For this algorithm and many other randomized algorithms, *no particular input elicits its worst-case behavior*. Even your worst enemy cannot produce a bad input array, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if the random-number generator produces an "unlucky" permutation.

For the hiring problem, the only change needed in the code is to randomly permute the array.

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  randomly permute the list of candidates
2  best = 0            // candidate 0 is a least-qualified dummy candidate
3  for i = 1 to n
4      interview candidate i
5      if candidate i is better than candidate best
6          best = i
7          hire candidate i
```

With this simple change, we have created a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

Lemma 5.3
The expected hiring cost of the procedure RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.

Proof After permuting the input array, we have achieved a situation identical to that of the probabilistic analysis of HIRE-ASSISTANT. ■

Comparing Lemmas 5.2 and 5.3 highlights the difference between probabilistic analysis and randomized algorithms. In Lemma 5.2, we make an assumption about the input. In Lemma 5.3, we make no such assumption, although randomizing the input takes some additional time. To remain consistent with our terminology, we couched Lemma 5.2 in terms of the average-case hiring cost and Lemma 5.3 in terms of the expected hiring cost. In the remainder of this section, we discuss some issues involved in randomly permuting inputs.

Randomly permuting arrays

Many randomized algorithms randomize the input by permuting the given input array. (There are other ways to use randomization.) Here, we shall discuss two methods for doing so. We assume that we are given an array A which, without loss of generality, contains the elements 1 through n. Our goal is to produce a random permutation of the array.

One common method is to assign each element $A[i]$ of the array a random priority $P[i]$, and then sort the elements of A according to these priorities. For example, if our initial array is $A = \langle 1, 2, 3, 4 \rangle$ and we choose random priorities $P = \langle 36, 3, 62, 19 \rangle$, we would produce an array $B = \langle 2, 4, 1, 3 \rangle$, since the second priority is the smallest, followed by the fourth, then the first, and finally the third. We call this procedure PERMUTE-BY-SORTING:

```
PERMUTE-BY-SORTING(A)
1  n = A.length
2  let P[1..n] be a new array
3  for i = 1 to n
4      P[i] = RANDOM(1, n^3)
5  sort A, using P as sort keys
```

Line 4 chooses a random number between 1 and n^3. We use a range of 1 to n^3 to make it likely that all the priorities in P are unique. (Exercise 5.3-5 asks you to prove that the probability that all entries are unique is at least $1 - 1/n$, and Exercise 5.3-6 asks how to implement the algorithm even if two or more priorities are identical.) Let us assume that all the priorities are unique.

The time-consuming step in this procedure is the sorting in line 5. As we shall see in Chapter 8, if we use a comparison sort, sorting takes $\Omega(n \lg n)$ time. We can achieve this lower bound, since we have seen that merge sort takes $\Theta(n \lg n)$ time. (We shall see other comparison sorts that take $\Theta(n \lg n)$ time in Part II. Exercise 8.3-4 asks you to solve the very similar problem of sorting numbers in the range 0 to $n^3 - 1$ in $O(n)$ time.) After sorting, if $P[i]$ is the jth smallest priority, then $A[i]$ lies in position j of the output. In this manner we obtain a permutation. It remains to prove that the procedure produces a ***uniform random permutation***, that is, that the procedure is equally likely to produce every permutation of the numbers 1 through n.

Lemma 5.4
Procedure PERMUTE-BY-SORTING produces a uniform random permutation of the input, assuming that all priorities are distinct.

Proof We start by considering the particular permutation in which each element $A[i]$ receives the ith smallest priority. We shall show that this permutation occurs with probability exactly $1/n!$. For $i = 1, 2, \ldots, n$, let E_i be the event that element $A[i]$ receives the ith smallest priority. Then we wish to compute the probability that for all i, event E_i occurs, which is

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\} .$$

Using Exercise C.2-5, this probability is equal to

$$\Pr\{E_1\} \cdot \Pr\{E_2 \mid E_1\} \cdot \Pr\{E_3 \mid E_2 \cap E_1\} \cdot \Pr\{E_4 \mid E_3 \cap E_2 \cap E_1\} \\ \cdots \Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\} \cdots \Pr\{E_n \mid E_{n-1} \cap \cdots \cap E_1\} .$$

We have that $\Pr\{E_1\} = 1/n$ because it is the probability that one priority chosen randomly out of a set of n is the smallest priority. Next, we observe that $\Pr\{E_2 \mid E_1\} = 1/(n-1)$ because given that element $A[1]$ has the smallest priority, each of the remaining $n-1$ elements has an equal chance of having the second smallest priority. In general, for $i = 2, 3, \ldots, n$, we have that $\Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\} = 1/(n-i+1)$, since, given that elements $A[1]$ through $A[i-1]$ have the $i-1$ smallest priorities (in order), each of the remaining $n-(i-1)$ elements has an equal chance of having the ith smallest priority. Thus, we have

$$\begin{aligned}\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\} &= \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) \\ &= \frac{1}{n!},\end{aligned}$$

and we have shown that the probability of obtaining the identity permutation is $1/n!$.

We can extend this proof to work for any permutation of priorities. Consider any fixed permutation $\sigma = \langle \sigma(1), \sigma(2), \ldots, \sigma(n) \rangle$ of the set $\{1, 2, \ldots, n\}$. Let us denote by r_i the rank of the priority assigned to element $A[i]$, where the element with the jth smallest priority has rank j. If we define E_i as the event in which element $A[i]$ receives the $\sigma(i)$th smallest priority, or $r_i = \sigma(i)$, the same proof still applies. Therefore, if we calculate the probability of obtaining any particular permutation, the calculation is identical to the one above, so that the probability of obtaining this permutation is also $1/n!$. ■

You might think that to prove that a permutation is a uniform random permutation, it suffices to show that, for each element $A[i]$, the probability that the element winds up in position j is $1/n$. Exercise 5.3-4 shows that this weaker condition is, in fact, insufficient.

A better method for generating a random permutation is to permute the given array in place. The procedure RANDOMIZE-IN-PLACE does so in $O(n)$ time. In its ith iteration, it chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$. Subsequent to the ith iteration, $A[i]$ is never altered.

```
RANDOMIZE-IN-PLACE(A)
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(i, n)]
```

We shall use a loop invariant to show that procedure RANDOMIZE-IN-PLACE produces a uniform random permutation. A ***k-permutation*** on a set of n elements is a sequence containing k of the n elements, with no repetitions. (See Appendix C.) There are $n!/(n-k)!$ such possible k-permutations.

Lemma 5.5
Procedure RANDOMIZE-IN-PLACE computes a uniform random permutation.

Proof We use the following loop invariant:

> Just prior to the ith iteration of the **for** loop of lines 2–3, for each possible $(i-1)$-permutation of the n elements, the subarray $A[1\,..\,i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Consider the situation just before the first loop iteration, so that $i = 1$. The loop invariant says that for each possible 0-permutation, the subarray $A[1\,..\,0]$ contains this 0-permutation with probability $(n-i+1)!/n! = n!/n! = 1$. The subarray $A[1\,..\,0]$ is an empty subarray, and a 0-permutation has no elements. Thus, $A[1\,..\,0]$ contains any 0-permutation with probability 1, and the loop invariant holds prior to the first iteration.

Maintenance: We assume that just before the ith iteration, each possible $(i-1)$-permutation appears in the subarray $A[1\,..\,i-1]$ with probability $(n-i+1)!/n!$, and we shall show that after the ith iteration, each possible i-permutation appears in the subarray $A[1\,..\,i]$ with probability $(n-i)!/n!$. Incrementing i for the next iteration then maintains the loop invariant.

Let us examine the ith iteration. Consider a particular i-permutation, and denote the elements in it by $\langle x_1, x_2, \ldots, x_i\rangle$. This permutation consists of an $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1}\rangle$ followed by the value x_i that the algorithm places in $A[i]$. Let E_1 denote the event in which the first $i-1$ iterations have created the particular $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1}\rangle$ in $A[1\,..\,i-1]$. By the loop invariant, $\Pr\{E_1\} = (n-i+1)!/n!$. Let E_2 be the event that ith iteration puts x_i in position $A[i]$. The i-permutation $\langle x_1, \ldots, x_i\rangle$ appears in $A[1\,..\,i]$ precisely when both E_1 and E_2 occur, and so we wish to compute $\Pr\{E_2 \cap E_1\}$. Using equation (C.14), we have

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \ .$$

The probability $\Pr\{E_2 \mid E_1\}$ equals $1/(n-i+1)$ because in line 3 the algorithm chooses x_i randomly from the $n-i+1$ values in positions $A[i\,..\,n]$. Thus, we have

$$\begin{aligned}\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\}\Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!} .\end{aligned}$$

Termination: At termination, $i = n + 1$, and we have that the subarray $A[1 \mathinner{..} n]$ is a given n-permutation with probability $(n-(n+1)+1)/n! = 0!/n! = 1/n!$.

Thus, RANDOMIZE-IN-PLACE produces a uniform random permutation. ■

A randomized algorithm is often the simplest and most efficient way to solve a problem. We shall use randomized algorithms occasionally throughout this book.

Exercises

5.3-1

Professor Marceau objects to the loop invariant used in the proof of Lemma 5.5. He questions whether it is true prior to the first iteration. He reasons that we could just as easily declare that an empty subarray contains no 0-permutations. Therefore, the probability that an empty subarray contains a 0-permutation should be 0, thus invalidating the loop invariant prior to the first iteration. Rewrite the procedure RANDOMIZE-IN-PLACE so that its associated loop invariant applies to a nonempty subarray prior to the first iteration, and modify the proof of Lemma 5.5 for your procedure.

5.3-2

Professor Kelp decides to write a procedure that produces at random any permutation besides the identity permutation. He proposes the following procedure:

```
PERMUTE-WITHOUT-IDENTITY($A$)
1  $n = A.length$
2  for $i = 1$ to $n - 1$
3      swap $A[i]$ with $A[$RANDOM$(i + 1, n)]$
```

Does this code do what Professor Kelp intends?

5.3-3

Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i \mathinner{..} n]$, we swapped it with a random element from anywhere in the array:

```
PERMUTE-WITH-ALL(A)
1  $n = A.length$
2  for $i = 1$ to $n$
3      swap $A[i]$ with $A[\text{RANDOM}(1, n)]$
```

Does this code produce a uniform random permutation? Why or why not?

5.3-4
Professor Armstrong suggests the following procedure for generating a uniform random permutation:

```
PERMUTE-BY-CYCLIC(A)
1  $n = A.length$
2  let $B[1..n]$ be a new array
3  $offset = \text{RANDOM}(1, n)$
4  for $i = 1$ to $n$
5      $dest = i + offset$
6      if $dest > n$
7          $dest = dest - n$
8      $B[dest] = A[i]$
9  return $B$
```

Show that each element $A[i]$ has a $1/n$ probability of winding up in any particular position in B. Then show that Professor Armstrong is mistaken by showing that the resulting permutation is not uniformly random.

5.3-5 ★
Prove that in the array P in procedure PERMUTE-BY-SORTING, the probability that all elements are unique is at least $1 - 1/n$.

5.3-6
Explain how to implement the algorithm PERMUTE-BY-SORTING to handle the case in which two or more priorities are identical. That is, your algorithm should produce a uniform random permutation, even if two or more priorities are identical.

5.3-7
Suppose we want to create a ***random sample*** of the set $\{1, 2, 3, \ldots, n\}$, that is, an m-element subset S, where $0 \le m \le n$, such that each m-subset is equally likely to be created. One way would be to set $A[i] = i$ for $i = 1, 2, 3, \ldots, n$, call RANDOMIZE-IN-PLACE(A), and then take just the first m array elements. This method would make n calls to the RANDOM procedure. If n is much larger than m, we can create a random sample with fewer calls to RANDOM. Show that

the following recursive procedure returns a random m-subset S of $\{1, 2, 3, \ldots, n\}$, in which each m-subset is equally likely, while making only m calls to RANDOM:

```
RANDOM-SAMPLE($m, n$)
1  if $m == 0$
2      return $\emptyset$
3  else $S$ = RANDOM-SAMPLE($m - 1, n - 1$)
4      $i$ = RANDOM($1, n$)
5      if $i \in S$
6          $S = S \cup \{n\}$
7      else $S = S \cup \{i\}$
8      return $S$
```

★ 5.4 Probabilistic analysis and further uses of indicator random variables

This advanced section further illustrates probabilistic analysis by way of four examples. The first determines the probability that in a room of k people, two of them share the same birthday. The second example examines what happens when we randomly toss balls into bins. The third investigates "streaks" of consecutive heads when we flip coins. The final example analyzes a variant of the hiring problem in which you have to make decisions without actually interviewing all the candidates.

5.4.1 The birthday paradox

Our first example is the ***birthday paradox***. How many people must there be in a room before there is a 50% chance that two of them were born on the same day of the year? The answer is surprisingly few. The paradox is that it is in fact far fewer than the number of days in a year, or even half the number of days in a year, as we shall see.

To answer this question, we index the people in the room with the integers $1, 2, \ldots, k$, where k is the number of people in the room. We ignore the issue of leap years and assume that all years have $n = 365$ days. For $i = 1, 2, \ldots, k$, let b_i be the day of the year on which person i's birthday falls, where $1 \le b_i \le n$. We also assume that birthdays are uniformly distributed across the n days of the year, so that $\Pr\{b_i = r\} = 1/n$ for $i = 1, 2, \ldots, k$ and $r = 1, 2, \ldots, n$.

The probability that two given people, say i and j, have matching birthdays depends on whether the random selection of birthdays is independent. We assume from now on that birthdays are independent, so that the probability that i's birthday and j's birthday both fall on day r is

$$\begin{aligned}\Pr\{b_i = r \text{ and } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= 1/n^2 .\end{aligned}$$

Thus, the probability that they both fall on the same day is

$$\begin{aligned}\Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ and } b_j = r\} \\ &= \sum_{r=1}^{n} (1/n^2) \\ &= 1/n . \qquad (5.6)\end{aligned}$$

More intuitively, once b_i is chosen, the probability that b_j is chosen to be the same day is $1/n$. Thus, the probability that i and j have the same birthday is the same as the probability that the birthday of one of them falls on a given day. Notice, however, that this coincidence depends on the assumption that the birthdays are independent.

We can analyze the probability of at least 2 out of k people having matching birthdays by looking at the complementary event. The probability that at least two of the birthdays match is 1 minus the probability that all the birthdays are different. The event that k people have distinct birthdays is

$$B_k = \bigcap_{i=1}^{k} A_i ,$$

where A_i is the event that person i's birthday is different from person j's for all $j < i$. Since we can write $B_k = A_k \cap B_{k-1}$, we obtain from equation (C.16) the recurrence

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} , \qquad (5.7)$$

where we take $\Pr\{B_1\} = \Pr\{A_1\} = 1$ as an initial condition. In other words, the probability that $b_1, b_2, \ldots, b_k$ are distinct birthdays is the probability that $b_1, b_2, \ldots, b_{k-1}$ are distinct birthdays times the probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k-1$, given that $b_1, b_2, \ldots, b_{k-1}$ are distinct.

If $b_1, b_2, \ldots, b_{k-1}$ are distinct, the conditional probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k-1$ is $\Pr\{A_k \mid B_{k-1}\} = (n-k+1)/n$, since out of the n days, $n-(k-1)$ days are not taken. We iteratively apply the recurrence (5.7) to obtain

$$
\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-2}\}\Pr\{A_{k-1} \mid B_{k-2}\}\Pr\{A_k \mid B_{k-1}\} \\
&\vdots \\
&= \Pr\{B_1\}\Pr\{A_2 \mid B_1\}\Pr\{A_3 \mid B_2\}\cdots\Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right) \\
&= 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right) .
\end{aligned}
$$

Inequality (3.12), $1 + x \le e^x$, gives us

$$
\begin{aligned}
\Pr\{B_k\} &\le e^{-1/n}e^{-2/n}\cdots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\le 1/2
\end{aligned}
$$

when $-k(k-1)/2n \le \ln(1/2)$. The probability that all k birthdays are distinct is at most $1/2$ when $k(k-1) \ge 2n \ln 2$ or, solving the quadratic equation, when $k \ge (1 + \sqrt{1 + (8 \ln 2)n})/2$. For $n = 365$, we must have $k \ge 23$. Thus, if at least 23 people are in a room, the probability is at least $1/2$ that at least two people have the same birthday. On Mars, a year is 669 Martian days long; it therefore takes 31 Martians to get the same effect.

An analysis using indicator random variables

We can use indicator random variables to provide a simpler but approximate analysis of the birthday paradox. For each pair (i, j) of the k people in the room, we define the indicator random variable X_{ij}, for $1 \le i < j \le k$, by

$$
\begin{aligned}
X_{ij} &= \mathrm{I}\{\text{person } i \text{ and person } j \text{ have the same birthday}\} \\
&= \begin{cases} 1 & \text{if person } i \text{ and person } j \text{ have the same birthday ,} \\ 0 & \text{otherwise .} \end{cases}
\end{aligned}
$$

By equation (5.6), the probability that two people have matching birthdays is $1/n$, and thus by Lemma 5.1, we have

$$
\begin{aligned}
\mathrm{E}[X_{ij}] &= \Pr\{\text{person } i \text{ and person } j \text{ have the same birthday}\} \\
&= 1/n .
\end{aligned}
$$

Letting X be the random variable that counts the number of pairs of individuals having the same birthday, we have

$$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij} \ .$$

Taking expectations of both sides and applying linearity of expectation, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] \\
&= \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathrm{E}[X_{ij}] \\
&= \binom{k}{2} \frac{1}{n} \\
&= \frac{k(k-1)}{2n} \ .
\end{aligned}$$

When $k(k-1) \geq 2n$, therefore, the expected number of pairs of people with the same birthday is at least 1. Thus, if we have at least $\sqrt{2n} + 1$ individuals in a room, we can expect at least two to have the same birthday. For $n = 365$, if $k = 28$, the expected number of pairs with the same birthday is $(28 \cdot 27)/(2 \cdot 365) \approx 1.0356$. Thus, with at least 28 people, we expect to find at least one matching pair of birthdays. On Mars, where a year is 669 Martian days long, we need at least 38 Martians.

The first analysis, which used only probabilities, determined the number of people required for the probability to exceed $1/2$ that a matching pair of birthdays exists, and the second analysis, which used indicator random variables, determined the number such that the expected number of matching birthdays is 1. Although the exact numbers of people differ for the two situations, they are the same asymptotically: $\Theta(\sqrt{n})$.

5.4.2 Balls and bins

Consider a process in which we randomly toss identical balls into b bins, numbered $1, 2, \ldots, b$. The tosses are independent, and on each toss the ball is equally likely to end up in any bin. The probability that a tossed ball lands in any given bin is $1/b$. Thus, the ball-tossing process is a sequence of Bernoulli trials (see Appendix C.4) with a probability $1/b$ of success, where success means that the ball falls in the given bin. This model is particularly useful for analyzing hashing (see Chapter 11), and we can answer a variety of interesting questions about the ball-tossing process. (Problem C-1 asks additional questions about balls and bins.)

How many balls fall in a given bin? The number of balls that fall in a given bin follows the binomial distribution $b(k; n, 1/b)$. If we toss n balls, equation (C.37) tells us that the expected number of balls that fall in the given bin is n/b.

How many balls must we toss, on the average, until a given bin contains a ball? The number of tosses until the given bin receives a ball follows the geometric distribution with probability $1/b$ and, by equation (C.32), the expected number of tosses until success is $1/(1/b) = b$.

How many balls must we toss until every bin contains at least one ball? Let us call a toss in which a ball falls into an empty bin a "hit." We want to know the expected number n of tosses required to get b hits.

Using the hits, we can partition the n tosses into stages. The ith stage consists of the tosses after the $(i-1)$st hit until the ith hit. The first stage consists of the first toss, since we are guaranteed to have a hit when all bins are empty. For each toss during the ith stage, $i-1$ bins contain balls and $b-i+1$ bins are empty. Thus, for each toss in the ith stage, the probability of obtaining a hit is $(b-i+1)/b$.

Let n_i denote the number of tosses in the ith stage. Thus, the number of tosses required to get b hits is $n = \sum_{i=1}^{b} n_i$. Each random variable n_i has a geometric distribution with probability of success $(b-i+1)/b$ and thus, by equation (C.32), we have

$$\mathrm{E}[n_i] = \frac{b}{b-i+1} .$$

By linearity of expectation, we have

$$\begin{aligned}
\mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} \mathrm{E}[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b \sum_{i=1}^{b} \frac{1}{i} \\
&= b(\ln b + O(1)) \quad \text{(by equation (A.7))} .
\end{aligned}$$

It therefore takes approximately $b \ln b$ tosses before we can expect that every bin has a ball. This problem is also known as the ***coupon collector's problem***, which says that a person trying to collect each of b different coupons expects to acquire approximately $b \ln b$ randomly obtained coupons in order to succeed.

5.4.3 Streaks

Suppose you flip a fair coin n times. What is the longest streak of consecutive heads that you expect to see? The answer is $\Theta(\lg n)$, as the following analysis shows.

We first prove that the expected length of the longest streak of heads is $O(\lg n)$. The probability that each coin flip is a head is $1/2$. Let A_{ik} be the event that a streak of heads of length at least k begins with the ith coin flip or, more precisely, the event that the k consecutive coin flips $i, i+1, \ldots, i+k-1$ yield only heads, where $1 \le k \le n$ and $1 \le i \le n-k+1$. Since coin flips are mutually independent, for any given event A_{ik}, the probability that all k flips are heads is

$$\Pr\{A_{ik}\} = 1/2^k \ . \tag{5.8}$$

For $k = 2\lceil \lg n \rceil$,

$$\begin{aligned} \Pr\{A_{i,2\lceil \lg n \rceil}\} &= 1/2^{2\lceil \lg n \rceil} \\ &\le 1/2^{2\lg n} \\ &= 1/n^2 \ , \end{aligned}$$

and thus the probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins in position i is quite small. There are at most $n - 2\lceil \lg n \rceil + 1$ positions where such a streak can begin. The probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins anywhere is therefore

$$\begin{aligned} \Pr\left\{ \bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil} \right\} &\le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} 1/n^2 \\ &< \sum_{i=1}^{n} 1/n^2 \\ &= 1/n \ , \end{aligned} \tag{5.9}$$

since by Boole's inequality (C.19), the probability of a union of events is at most the sum of the probabilities of the individual events. (Note that Boole's inequality holds even for events such as these that are not independent.)

We now use inequality (5.9) to bound the length of the longest streak. For $j = 0, 1, 2, \ldots, n$, let L_j be the event that the longest streak of heads has length exactly j, and let L be the length of the longest streak. By the definition of expected value, we have

$$\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\{L_j\} \ . \tag{5.10}$$

We could try to evaluate this sum using upper bounds on each $\Pr\{L_j\}$ similar to those computed in inequality (5.9). Unfortunately, this method would yield weak bounds. We can use some intuition gained by the above analysis to obtain a good bound, however. Informally, we observe that for no individual term in the summation in equation (5.10) are both the factors j and $\Pr\{L_j\}$ large. Why? When $j \geq 2\lceil\lg n\rceil$, then $\Pr\{L_j\}$ is very small, and when $j < 2\lceil\lg n\rceil$, then j is fairly small. More formally, we note that the events L_j for $j = 0, 1, \ldots, n$ are disjoint, and so the probability that a streak of heads of length at least $2\lceil\lg n\rceil$ begins anywhere is $\sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\}$. By inequality (5.9), we have $\sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\} < 1/n$. Also, noting that $\sum_{j=0}^{n} \Pr\{L_j\} = 1$, we have that $\sum_{j=0}^{2\lceil\lg n\rceil-1} \Pr\{L_j\} \leq 1$. Thus, we obtain

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil\lg n\rceil-1} j \Pr\{L_j\} + \sum_{j=2\lceil\lg n\rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil\lg n\rceil-1} (2\lceil\lg n\rceil) \Pr\{L_j\} + \sum_{j=2\lceil\lg n\rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil\lg n\rceil \sum_{j=0}^{2\lceil\lg n\rceil-1} \Pr\{L_j\} + n \sum_{j=2\lceil\lg n\rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil\lg n\rceil \cdot 1 + n \cdot (1/n) \\
&= O(\lg n) .
\end{aligned}$$

The probability that a streak of heads exceeds $r\lceil\lg n\rceil$ flips diminishes quickly with r. For $r \geq 1$, the probability that a streak of at least $r\lceil\lg n\rceil$ heads starts in position i is

$$\begin{aligned}
\Pr\{A_{i,r\lceil\lg n\rceil}\} &= 1/2^{r\lceil\lg n\rceil} \\
&\leq 1/n^r .
\end{aligned}$$

Thus, the probability is at most $n/n^r = 1/n^{r-1}$ that the longest streak is at least $r\lceil\lg n\rceil$, or equivalently, the probability is at least $1 - 1/n^{r-1}$ that the longest streak has length less than $r\lceil\lg n\rceil$.

As an example, for $n = 1000$ coin flips, the probability of having a streak of at least $2\lceil\lg n\rceil = 20$ heads is at most $1/n = 1/1000$. The chance of having a streak longer than $3\lceil\lg n\rceil = 30$ heads is at most $1/n^2 = 1/1{,}000{,}000$.

We now prove a complementary lower bound: the expected length of the longest streak of heads in n coin flips is $\Omega(\lg n)$. To prove this bound, we look for streaks of length s by partitioning the n flips into approximately n/s groups of s flips each. If we choose $s = \lfloor (\lg n)/2 \rfloor$, we can show that it is likely that at least one of these groups comes up all heads, and hence it is likely that the longest streak has length at least $s = \Omega(\lg n)$. We then show that the longest streak has expected length $\Omega(\lg n)$.

We partition the n coin flips into at least $\lfloor n / \lfloor (\lg n)/2 \rfloor \rfloor$ groups of $\lfloor (\lg n)/2 \rfloor$ consecutive flips, and we bound the probability that no group comes up all heads. By equation (5.8), the probability that the group starting in position i comes up all heads is

$$\begin{aligned} \Pr\{A_{i,\lfloor (\lg n)/2 \rfloor}\} &= 1/2^{\lfloor (\lg n)/2 \rfloor} \\ &\geq 1/\sqrt{n} \;. \end{aligned}$$

The probability that a streak of heads of length at least $\lfloor (\lg n)/2 \rfloor$ does not begin in position i is therefore at most $1 - 1/\sqrt{n}$. Since the $\lfloor n / \lfloor (\lg n)/2 \rfloor \rfloor$ groups are formed from mutually exclusive, independent coin flips, the probability that every one of these groups *fails* to be a streak of length $\lfloor (\lg n)/2 \rfloor$ is at most

$$\begin{aligned} \left(1 - 1/\sqrt{n}\right)^{\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor} &\leq \left(1 - 1/\sqrt{n}\right)^{n/\lfloor (\lg n)/2 \rfloor - 1} \\ &\leq \left(1 - 1/\sqrt{n}\right)^{2n/\lg n - 1} \\ &\leq e^{-(2n/\lg n - 1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O(1/n) \;. \end{aligned}$$

For this argument, we used inequality (3.12), $1 + x \leq e^x$, and the fact, which you might want to verify, that $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$ for sufficiently large n.

Thus, the probability that the longest streak exceeds $\lfloor (\lg n)/2 \rfloor$ is

$$\sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \Pr\{L_j\} \geq 1 - O(1/n) \;. \tag{5.11}$$

We can now calculate a lower bound on the expected length of the longest streak, beginning with equation (5.10) and proceeding in a manner similar to our analysis of the upper bound:

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} j \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} j \Pr\{L_j\} \\
&\geq \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \lfloor(\lg n)/2\rfloor \Pr\{L_j\} \\
&= 0 \cdot \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} \Pr\{L_j\} + \lfloor(\lg n)/2\rfloor \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \Pr\{L_j\} \\
&\geq 0 + \lfloor(\lg n)/2\rfloor\,(1 - O(1/n)) \qquad \text{(by inequality (5.11))} \\
&= \Omega(\lg n)\ .
\end{aligned}$$

As with the birthday paradox, we can obtain a simpler but approximate analysis using indicator random variables. We let $X_{ik} = \mathrm{I}\{A_{ik}\}$ be the indicator random variable associated with a streak of heads of length at least k beginning with the ith coin flip. To count the total number of such streaks, we define

$$X = \sum_{i=1}^{n-k+1} X_{ik}\ .$$

Taking expectations and using linearity of expectation, we have

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
&= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\
&= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\
&= \sum_{i=1}^{n-k+1} 1/2^k \\
&= \frac{n-k+1}{2^k}\ .
\end{aligned}$$

By plugging in various values for k, we can calculate the expected number of streaks of length k. If this number is large (much greater than 1), then we expect many streaks of length k to occur and the probability that one occurs is high. If this number is small (much less than 1), then we expect few streaks of length k to occur and the probability that one occurs is low. If $k = c \lg n$, for some positive constant c, we obtain

$$\begin{aligned} \mathrm{E}[X] &= \frac{n - c\lg n + 1}{2^{c \lg n}} \\ &= \frac{n - c\lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c\lg n - 1)/n}{n^{c-1}} \\ &= \Theta(1/n^{c-1}) . \end{aligned}$$

If c is large, the expected number of streaks of length $c \lg n$ is small, and we conclude that they are unlikely to occur. On the other hand, if $c = 1/2$, then we obtain $\mathrm{E}[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, and we expect that there are a large number of streaks of length $(1/2)\lg n$. Therefore, one streak of such a length is likely to occur. From these rough estimates alone, we can conclude that the expected length of the longest streak is $\Theta(\lg n)$.

5.4.4 The on-line hiring problem

As a final example, we consider a variant of the hiring problem. Suppose now that we do not wish to interview all the candidates in order to find the best one. We also do not wish to hire and fire as we find better and better applicants. Instead, we are willing to settle for a candidate who is close to the best, in exchange for hiring exactly once. We must obey one company requirement: after each interview we must either immediately offer the position to the applicant or immediately reject the applicant. What is the trade-off between minimizing the amount of interviewing and maximizing the quality of the candidate hired?

We can model this problem in the following way. After meeting an applicant, we are able to give each one a score; let $score(i)$ denote the score we give to the ith applicant, and assume that no two applicants receive the same score. After we have seen j applicants, we know which of the j has the highest score, but we do not know whether any of the remaining $n-j$ applicants will receive a higher score. We decide to adopt the strategy of selecting a positive integer $k < n$, interviewing and then rejecting the first k applicants, and hiring the first applicant thereafter who has a higher score than all preceding applicants. If it turns out that the best-qualified applicant was among the first k interviewed, then we hire the nth applicant. We formalize this strategy in the procedure ON-LINE-MAXIMUM(k, n), which returns the index of the candidate we wish to hire.

```
ON-LINE-MAXIMUM(k, n)
1  bestscore = −∞
2  for i = 1 to k
3      if score(i) > bestscore
4          bestscore = score(i)
5  for i = k + 1 to n
6      if score(i) > bestscore
7          return i
8  return n
```

We wish to determine, for each possible value of k, the probability that we hire the most qualified applicant. We then choose the best possible k, and implement the strategy with that value. For the moment, assume that k is fixed. Let $M(j) = \max_{1 \le i \le j} \{score(i)\}$ denote the maximum score among applicants 1 through j. Let S be the event that we succeed in choosing the best-qualified applicant, and let S_i be the event that we succeed when the best-qualified applicant is the ith one interviewed. Since the various S_i are disjoint, we have that $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Noting that we never succeed when the best-qualified applicant is one of the first k, we have that $\Pr\{S_i\} = 0$ for $i = 1, 2, \ldots, k$. Thus, we obtain

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} \; . \tag{5.12}$$

We now compute $\Pr\{S_i\}$. In order to succeed when the best-qualified applicant is the ith one, two things must happen. First, the best-qualified applicant must be in position i, an event which we denote by B_i. Second, the algorithm must not select any of the applicants in positions $k+1$ through $i-1$, which happens only if, for each j such that $k+1 \le j \le i-1$, we find that $score(j) < bestscore$ in line 6. (Because scores are unique, we can ignore the possibility of $score(j) = bestscore$.) In other words, all of the values $score(k+1)$ through $score(i-1)$ must be less than $M(k)$; if any are greater than $M(k)$, we instead return the index of the first one that is greater. We use O_i to denote the event that none of the applicants in position $k+1$ through $i-1$ are chosen. Fortunately, the two events B_i and O_i are independent. The event O_i depends only on the relative ordering of the values in positions 1 through $i-1$, whereas B_i depends only on whether the value in position i is greater than the values in all other positions. The ordering of the values in positions 1 through $i-1$ does not affect whether the value in position i is greater than all of them, and the value in position i does not affect the ordering of the values in positions 1 through $i-1$. Thus we can apply equation (C.15) to obtain

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\} \ .$$

The probability $\Pr\{B_i\}$ is clearly $1/n$, since the maximum is equally likely to be in any one of the n positions. For event O_i to occur, the maximum value in positions 1 through $i-1$, which is equally likely to be in any of these $i-1$ positions, must be in one of the first k positions. Consequently, $\Pr\{O_i\} = k/(i-1)$ and $\Pr\{S_i\} = k/(n(i-1))$. Using equation (5.12), we have

$$\begin{aligned}
\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\
&= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \\
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} \ .
\end{aligned}$$

We approximate by integrals to bound this summation from above and below. By the inequalities (A.12), we have

$$\int_k^n \frac{1}{x}\,dx \le \sum_{i=k}^{n-1} \frac{1}{i} \le \int_{k-1}^{n-1} \frac{1}{x}\,dx \ .$$

Evaluating these definite integrals gives us the bounds

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1)) \ ,$$

which provide a rather tight bound for $\Pr\{S\}$. Because we wish to maximize our probability of success, let us focus on choosing the value of k that maximizes the lower bound on $\Pr\{S\}$. (Besides, the lower-bound expression is easier to maximize than the upper-bound expression.) Differentiating the expression $(k/n)(\ln n - \ln k)$ with respect to k, we obtain

$$\frac{1}{n}(\ln n - \ln k - 1) \ .$$

Setting this derivative equal to 0, we see that we maximize the lower bound on the probability when $\ln k = \ln n - 1 = \ln(n/e)$ or, equivalently, when $k = n/e$. Thus, if we implement our strategy with $k = n/e$, we succeed in hiring our best-qualified applicant with probability at least $1/e$.

Exercises

5.4-1

How many people must there be in a room before the probability that someone has the same birthday as you do is at least $1/2$? How many people must there be before the probability that at least two people have a birthday on July 4 is greater than $1/2$?

5.4-2

Suppose that we toss balls into b bins until some bin contains two balls. Each toss is independent, and each ball is equally likely to end up in any bin. What is the expected number of ball tosses?

5.4-3 ★

For the analysis of the birthday paradox, is it important that the birthdays be mutually independent, or is pairwise independence sufficient? Justify your answer.

5.4-4 ★

How many people should be invited to a party in order to make it likely that there are *three* people with the same birthday?

5.4-5 ★

What is the probability that a k-string over a set of size n forms a k-permutation? How does this question relate to the birthday paradox?

5.4-6 ★

Suppose that n balls are tossed into n bins, where each toss is independent and the ball is equally likely to end up in any bin. What is the expected number of empty bins? What is the expected number of bins with exactly one ball?

5.4-7 ★

Sharpen the lower bound on streak length by showing that in n flips of a fair coin, the probability is less than $1/n$ that no streak longer than $\lg n - 2 \lg \lg n$ consecutive heads occurs.

Problems

5-1 Probabilistic counting

With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris's ***probabilistic counting***, we can count up to a much larger value at the expense of some loss of precision.

We let a counter value of i represent a count of n_i for $i = 0, 1, \ldots, 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The INCREMENT operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the INCREMENT operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$, and it leaves the counter unchanged with probability $1 - 1/(n_{i+1} - n_i)$.

If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number—see Section 3.2).

For this problem, assume that n_{2^b-1} is large enough that the probability of an overflow error is negligible.

a. Show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n.

b. The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

5-2 Searching an unsorted array

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If $A[i] = x$, then we terminate; otherwise, we continue the search by picking a new random index into A. We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A. Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

b. Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates?

c. Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

d. Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we have checked all elements of A and RANDOM-SEARCH terminates?

Now consider a deterministic linear search algorithm, which we refer to as DETERMINISTIC-SEARCH. Specifically, the algorithm searches A for x in order, considering $A[1], A[2], A[3], \ldots, A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

e. Suppose that there is exactly one index i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

f. Generalizing your solution to part (e), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH? Your answer should be a function of n and k.

g. Suppose that there are no indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

Finally, consider a randomized algorithm SCRAMBLE-SEARCH that works by first randomly permuting the input array and then running the deterministic linear search given above on the resulting permuted array.

h. Letting k be the number of indices i such that $A[i] = x$, give the worst-case and expected running times of SCRAMBLE-SEARCH for the cases in which $k = 0$ and $k = 1$. Generalize your solution to handle the case in which $k \geq 1$.

i. Which of the three searching algorithms would you use? Explain your answer.

Chapter notes

Bollobás [53], Hofri [174], and Spencer [321] contain a wealth of advanced probabilistic techniques. The advantages of randomized algorithms are discussed and surveyed by Karp [200] and Rabin [288]. The textbook by Motwani and Raghavan [262] gives an extensive treatment of randomized algorithms.

Several variants of the hiring problem have been widely studied. These problems are more commonly referred to as "secretary problems." An example of work in this area is the paper by Ajtai, Meggido, and Waarts [11].

Red-Balck Trees

13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can support any of the basic dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE—in $O(h)$ time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

13.1 Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL (see page 238). For a red-black tree T, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its *color* attribute is BLACK, and its other attributes—p, *left*, *right*, and *key*—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

We use the sentinel so that we can treat a NIL child of a node x as an ordinary node whose parent is x. Although we instead could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined, that approach would waste space. Instead, we use the one sentinel $T.nil$ to represent all the NILs—all leaves and the root's parent. The values of the attributes p, *left*, *right*, and *key* of the sentinel are immaterial, although we may set them during the course of a procedure for our convenience.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. In the remainder of this chapter, we omit the leaves when we draw red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted $\text{bh}(x)$. By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the black-height of its root.

The following lemma shows why red-black trees make good search trees.

Lemma 13.1
A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Proof We start by showing that the subtree rooted at any node x contains at least $2^{\text{bh}(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x indeed contains at least $2^{\text{bh}(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $\text{bh}(x)$ or $\text{bh}(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{\text{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not

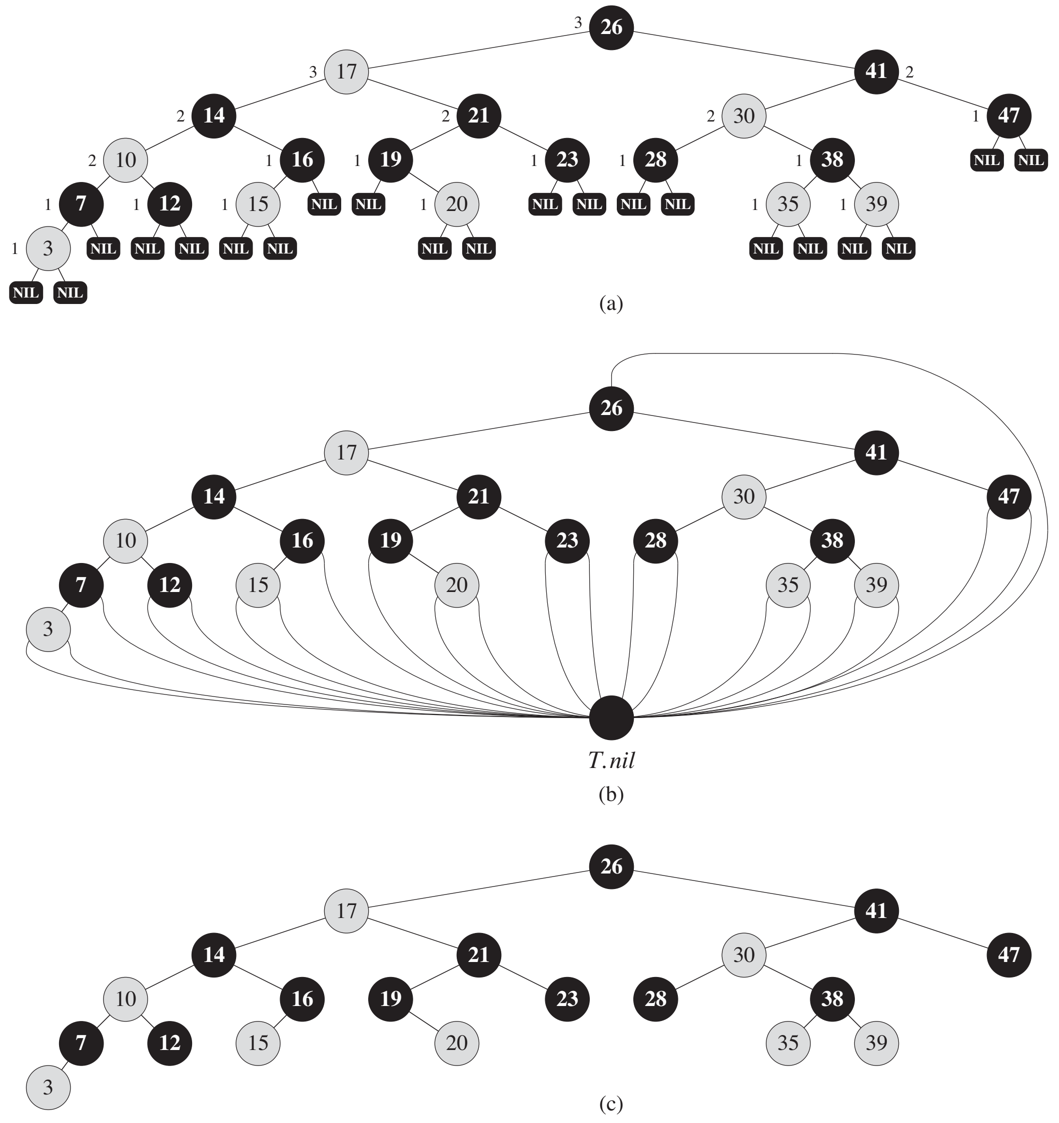


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus,

$$n \geq 2^{h/2} - 1 .$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n+1) \geq h/2$, or $h \leq 2\lg(n+1)$. ■

As an immediate consequence of this lemma, we can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg n)$ time on red-black trees, since each can run in $O(h)$ time on a binary search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes is a binary search tree with height $O(\lg n)$. (Of course, references to NIL in the algorithms of Chapter 12 would have to be replaced by $T.nil$.) Although the algorithms TREE-INSERT and TREE-DELETE from Chapter 12 run in $O(\lg n)$ time when given a red-black tree as input, they do not directly support the dynamic-set operations INSERT and DELETE, since they do not guarantee that the modified binary search tree will be a red-black tree. We shall see in Sections 13.3 and 13.4, however, how to support these two operations in $O(\lg n)$ time.

Exercises

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

13.1-3

Let us define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T, is the resulting tree a red-black tree?

13.1-4

Suppose that we "absorb" every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

13.1-5
Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

13.1-6
What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

13.1-7
Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1. To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through ***rotation***, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. When we do a left rotation on a node x, we assume that its right child y is not $T.nil$; x may be any node in the tree whose right child is not $T.nil$. The left rotation "pivots" around the link from x to y. It makes y the new root of the subtree, with x as y's left child and y's left child as x's right child.

The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$.

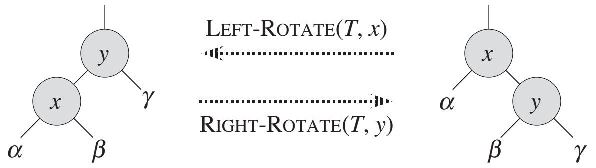


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

```
LEFT-ROTATE(T, x)
 1  y = x.right                 // set y
 2  x.right = y.left            // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                   // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x                  // put x on y's left
12  x.p = y
```

Figure 13.3 shows an example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation; all other attributes in a node remain the same.

Exercises

13.2-1
Write pseudocode for RIGHT-ROTATE.

13.2-2
Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

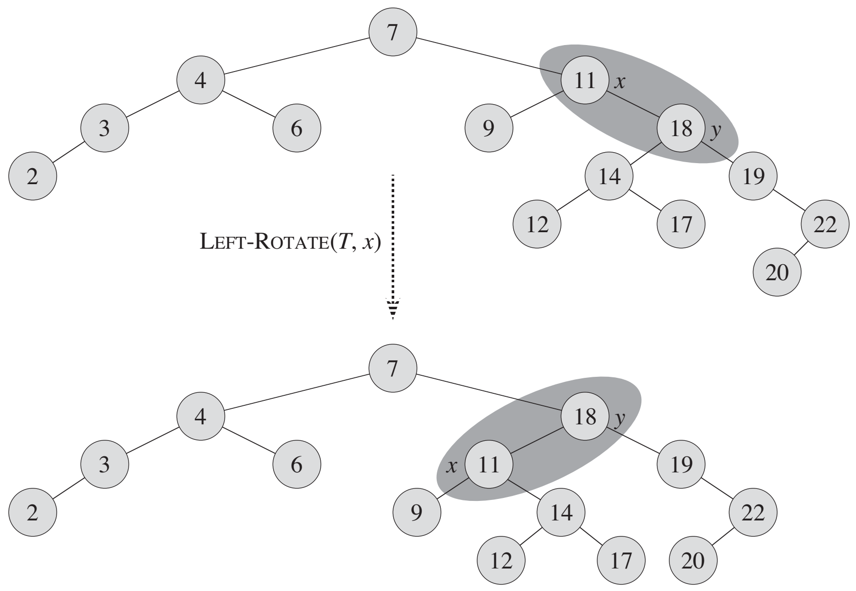


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-3

Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the left tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

13.2-4

Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. (*Hint:* First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

13.2-5 ★

We say that a binary search tree T_1 can be ***right-converted*** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

13.3 Insertion

We can insert a node into an n-node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the TREE-INSERT procedure (Section 12.3) to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. (Exercise 13.3-1 asks you to explain why we choose to make node z red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT(T, z) inserts node z, whose *key* is assumed to have already been filled in, into the red-black tree T.

```
RB-INSERT(T, z)
 1  $y = T.nil$
 2  $x = T.root$
 3  while $x \neq T.nil$
 4      $y = x$
 5      if $z.key < x.key$
 6          $x = x.left$
 7      else $x = x.right$
 8  $z.p = y$
 9  if $y == T.nil$
10      $T.root = z$
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
14  $z.left = T.nil$
15  $z.right = T.nil$
16  $z.color$ = RED
17  RB-INSERT-FIXUP$(T, z)$
```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by $T.nil$. Second, we set $z.left$ and $z.right$ to $T.nil$ in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color z red in line 16. Fourth, because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in line 17 of RB-INSERT to restore the red-black properties.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                          // case 1
 6              y.color = BLACK                            // case 1
 7              z.p.p.color = RED                          // case 1
 8              z = z.p.p                                  // case 1
 9          else if z == z.p.right
10                  z = z.p                                // case 2
11                  LEFT-ROTATE(T, z)                      // case 2
12              z.p.color = BLACK                          // case 3
13              z.p.p.color = RED                          // case 3
14              RIGHT-ROTATE(T, z.p.p)                     // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps. First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node z is inserted and colored red. Second, we shall examine the overall goal of the **while** loop in lines 1–15. Finally, we shall explore each of the three cases[1] within the **while** loop's body and see how they accomplish the goal. Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree.

Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel *T.nil*. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations are due to z being colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red. Figure 13.4(a) shows a violation of property 4 after the node z has been inserted.

[1] Case 2 falls through into case 3, and so these two cases are not mutually exclusive.

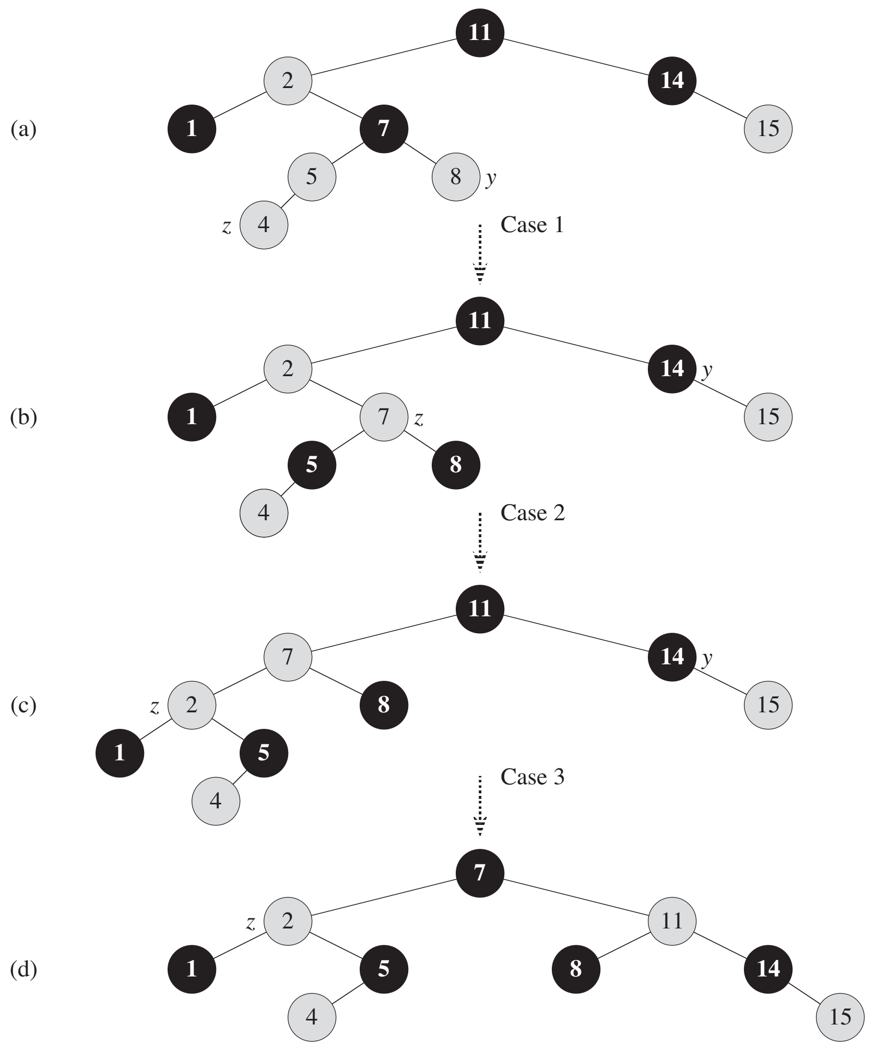


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. We perform a left rotation, and the tree that results is shown in **(c)**. Now, z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

The **while** loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

a. Node z is red.

b. If $z.p$ is the root, then $z.p$ is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we use along the way to understand situations in the code. Because we'll be focusing on node z and nodes near it in the tree, it helps to know from part (a) that z is red. We shall use part (b) to show that the node $z.p.p$ exists when we reference it in lines 2, 3, 7, 8, 13, and 14.

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

We start with the initialization and termination arguments. Then, as we examine how the body of the loop works in more detail, we shall argue that the loop maintains the invariant upon each iteration. Along the way, we shall also demonstrate that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or we perform some rotations and then the loop terminates.

Initialization: Prior to the first iteration of the loop, we started with a red-black tree with no violations, and we added a red node z. We show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, z is the red node that was added.

b. If $z.p$ is the root, then $z.p$ started out black and did not change prior to the call of RB-INSERT-FIXUP.

c. We have already seen that properties 1, 3, and 5 hold when RB-INSERT-FIXUP is called.

If the tree violates property 2, then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, the tree does not also violate property 4. Thus, this violation of property 2 is the only violation of red-black properties in the entire tree.

If the tree violates property 4, then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the violation must be because both z and $z.p$ are red. Moreover, the tree violates no other red-black properties.

Termination: When the loop terminates, it does so because $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) Thus, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 16 restores this property, too, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Maintenance: We actually need to consider six cases in the **while** loop, but three of them are symmetric to the other three, depending on whether line 2 determines z's parent $z.p$ to be a left child or a right child of z's grandparent $z.p.p$. We have given the code only for the situation in which $z.p$ is a left child. The node $z.p.p$ exists, since by part (b) of the loop invariant, if $z.p$ is the root, then $z.p$ is black. Since we enter a loop iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

We distinguish case 1 from cases 2 and 3 by the color of z's parent's sibling, or "uncle." Line 3 makes y point to z's uncle $z.p.p.right$, and line 4 tests y's color. If y is red, then we execute case 1. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent $z.p.p$ is black, since its parent $z.p$ is red, and property 4 is violated only between z and $z.p$.

Case 1: z's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because $z.p.p$ is black, we can color both $z.p$ and y black, thereby fixing the problem of z and $z.p$ both being red, and we can color $z.p.p$ red, thereby maintaining property 5. We then repeat the **while** loop with $z.p.p$ as the new node z. The pointer z moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and $z' = z.p.p$ to denote the node that will be called node z at the test in line 1 upon the next iteration.

a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

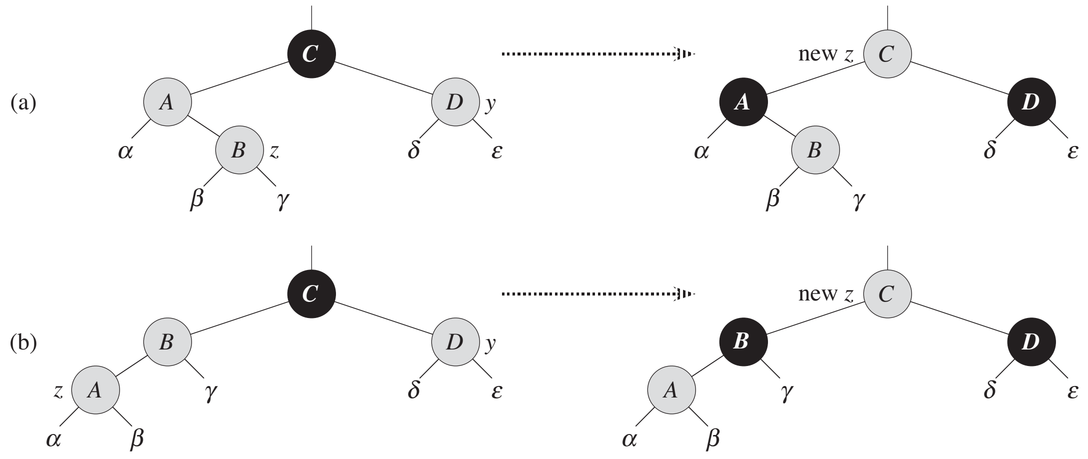


Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z and its parent $z.p$ are both red. We take the same action whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $z.p.p$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.

If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left $z'.p$ alone. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4 between z' and $z'.p$.

Case 2: z's uncle y is black and z is a right child
Case 3: z's uncle y is black and z is a left child

In cases 2 and 3, the color of z's uncle y is black. We distinguish the two cases according to whether z is a right or left child of $z.p$. Lines 10–11 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node z is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12–14), in which node z is a left child. Because

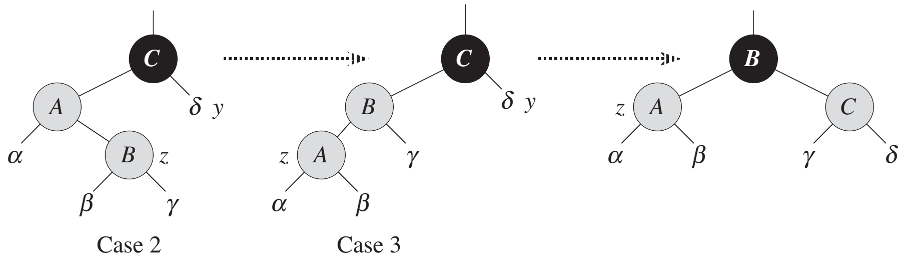


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

both z and $z.p$ are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2, z's uncle y is black, since otherwise we would have executed case 1. Additionally, the node $z.p.p$ exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 10 and then down one level in line 11, the identity of $z.p.p$ remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The **while** loop does not iterate another time, since $z.p$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $z.p$ will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

   Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

   Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Having shown that each iteration of the loop maintains the invariant, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, lines 1–16 of RB-INSERT take $O(\lg n)$ time. In RB-INSERT-FIXUP, the **while** loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Moreover, it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Exercises

13.3-1
In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z's color to black?

13.3-2
Show the red-black trees that result after successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree.

13.3-3
Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon$ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

13.3-4
Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

13.3-5
Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

13.3-6
Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

13.4 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node takes time $O(\lg n)$. Deleting a node from a red-black tree is a bit more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First, line 1 references the sentinel *T.nil* instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: we can assign to $v.p$ even if v points to the sentinel. In fact, we shall exploit the ability to assign to $v.p$ when $v = T.nil$.

The procedure RB-DELETE is like the TREE-DELETE procedure, but with additional lines of pseudocode. Some of the additional lines keep track of a node y that might cause violations of the red-black properties. When we want to delete node z and z has fewer than two children, then z is removed from the tree, and we want y to be z. When z has two children, then y should be z's successor, and y moves into z's position in the tree. We also remember y's color before it is removed from or moved within the tree, and we keep track of the node x that moves into y's original position in the tree, because node x might also cause violations of the red-black properties. After deleting node z, RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by $T.nil$ and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

Here are the other differences between the two procedures:

- We maintain node y as the node either removed from the tree or moved within the tree. Line 1 sets y to point to node z when z has fewer than two children and is therefore removed. When z has two children, line 9 sets y to point to z's successor, just as in TREE-DELETE, and y will move into z's position in the tree.
- Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y. When z has two children, then $y \neq z$ and node y moves into node z's original position in the red-black tree; line 20 gives y the same color as z. We need to save y's original color in order to test it at the

end of RB-DELETE; if it was black, then removing or moving y could cause violations of the red-black properties.

- As discussed, we keep track of the node x that moves into node y's original position. The assignments in lines 4, 7, and 11 set x to point to either y's only child or, if y has no children, the sentinel $T.nil$. (Recall from Section 12.3 that y has no left child.)
- Since node x moves into node y's original position, the attribute $x.p$ is always set to point to the original position in the tree of y's parent, even if x is, in fact, the sentinel $T.nil$. Unless z is y's original parent (which occurs only when z has two children and its successor y is z's right child), the assignment to $x.p$ takes place in line 6 of RB-TRANSPLANT. (Observe that when RB-TRANSPLANT is called in lines 5, 8, or 14, the second parameter passed is the same as x.)

  When y's original parent is z, however, we do not want $x.p$ to point to y's original parent, since we are removing that node from the tree. Because node y will move up to take z's position in the tree, setting $x.p$ to y in line 13 causes $x.p$ to point to the original position of y's parent, even if $x = T.nil$.
- Finally, if node y was black, we might have introduced one or more violations of the red-black properties, and so we call RB-DELETE-FIXUP in line 22 to restore the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

  1. No black-heights in the tree have changed.
  2. No red nodes have been made adjacent. Because y takes z's place in the tree, along with z's color, we cannot have two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. If y is red, then x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
  3. Since y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if y had been the root and a red child of y becomes the new root, we have violated property 2. Second, if both x and $x.p$ are red, then we have violated property 4. Third, moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct the violation of property 5 by saying that node x, now occupying y's original position, has an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds. When we remove or move the black node y, we "push" its blackness onto node x. The problem is that now node x is neither red nor black, thereby violating property 1. Instead, node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the *color* attribute.

We can now see the procedure RB-DELETE-FIXUP and examine how it restores the red-black properties to the search tree.

```
RB-DELETE-FIXUP$(T, x)$
 1  while $x \neq T.root$ and $x.color$ == BLACK
 2      if $x$ == $x.p.left$
 3          $w = x.p.right$
 4          if $w.color$ == RED
 5              $w.color$ = BLACK                                  // case 1
 6              $x.p.color$ = RED                                  // case 1
 7              LEFT-ROTATE$(T, x.p)$                              // case 1
 8              $w = x.p.right$                                    // case 1
 9          if $w.left.color$ == BLACK and $w.right.color$ == BLACK
10              $w.color$ = RED                                    // case 2
11              $x = x.p$                                          // case 2
12          else if $w.right.color$ == BLACK
13                  $w.left.color$ = BLACK                         // case 3
14                  $w.color$ = RED                                // case 3
15                  RIGHT-ROTATE$(T, w)$                           // case 3
16                  $w = x.p.right$                                // case 3
17              $w.color = x.p.color$                              // case 4
18              $x.p.color$ = BLACK                                // case 4
19              $w.right.color$ = BLACK                            // case 4
20              LEFT-ROTATE$(T, x.p)$                              // case 4
21              $x = T.root$                                       // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  $x.color$ = BLACK
```

The procedure RB-DELETE-FIXUP restores properties 1, 2, and 4. Exercises 13.4-1 and 13.4-2 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we shall focus on property 1. The goal of the **while** loop in lines 1–22 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case we color x (singly) black in line 23;
2. x points to the root, in which case we simply "remove" the extra black; or
3. having performed suitable rotations and recolorings, we exit the loop.

Within the **while** loop, x always points to a nonroot doubly black node. We determine in line 2 whether x is a left child or a right child of its parent $x.p$. (We have given the code for the situation in which x is a left child; the situation in which x is a right child—line 22—is symmetric.) We maintain a pointer w to the sibling of x. Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x.

The four cases[2] in the code appear in Figure 13.7. Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including x's extra black) from (and including) the root of the subtree shown to each of the subtrees $\alpha, \beta, \ldots, \zeta$. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to either subtree α or β is 3, both before and after the transformation. (Again, remember that node x adds an extra black.) Similarly, the number of black nodes from the root to any of γ, δ, ε, and ζ is 2, both before and after the transformation. In Figure 13.7(b), the counting must involve the value c of the *color* attribute of the root of the subtree shown, which can be either RED or BLACK. If we define count(RED) $= 0$ and count(BLACK) $= 1$, then the number of black nodes from the root to α is $2 +$ count(c), both before and after the transformation. In this case, after the transformation, the new node x has *color* attribute c, but this node is really either red-and-black (if $c =$ RED) or doubly black (if $c =$ BLACK). You can verify the other cases similarly (see Exercise 13.4-5).

Case 1: x's sibling w is red

Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.7(a)) occurs when node w, the sibling of node x, is red. Since w must have black children, we can switch the colors of w and $x.p$ and then perform a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black; they are distinguished by the colors of w's children.

[2] As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.

Case 2: x's sibling w is black, and both of w's children are black
In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.7(b)), both of w's children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, we would like to add an extra black to $x.p$, which was originally either red or black. We do so by repeating the **while** loop with $x.p$ as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the *color* attribute of the new node x is RED, and the loop terminates when it tests the loop condition. We then color the new node x (singly) black in line 23.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black
Case 3 (lines 13–16 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child $w.\mathit{left}$ and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.

Case 4: x's sibling w is black, and w's right child is red
Case 4 (lines 17–21 and Figure 13.7(d)) occurs when node x's sibling w is black and w's right child is red. By making some color changes and performing a left rotation on $x.p$, we can remove the extra black on x, making it singly black, without violating any of the red-black properties. Setting x to be the root causes the **while** loop to terminate when it tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

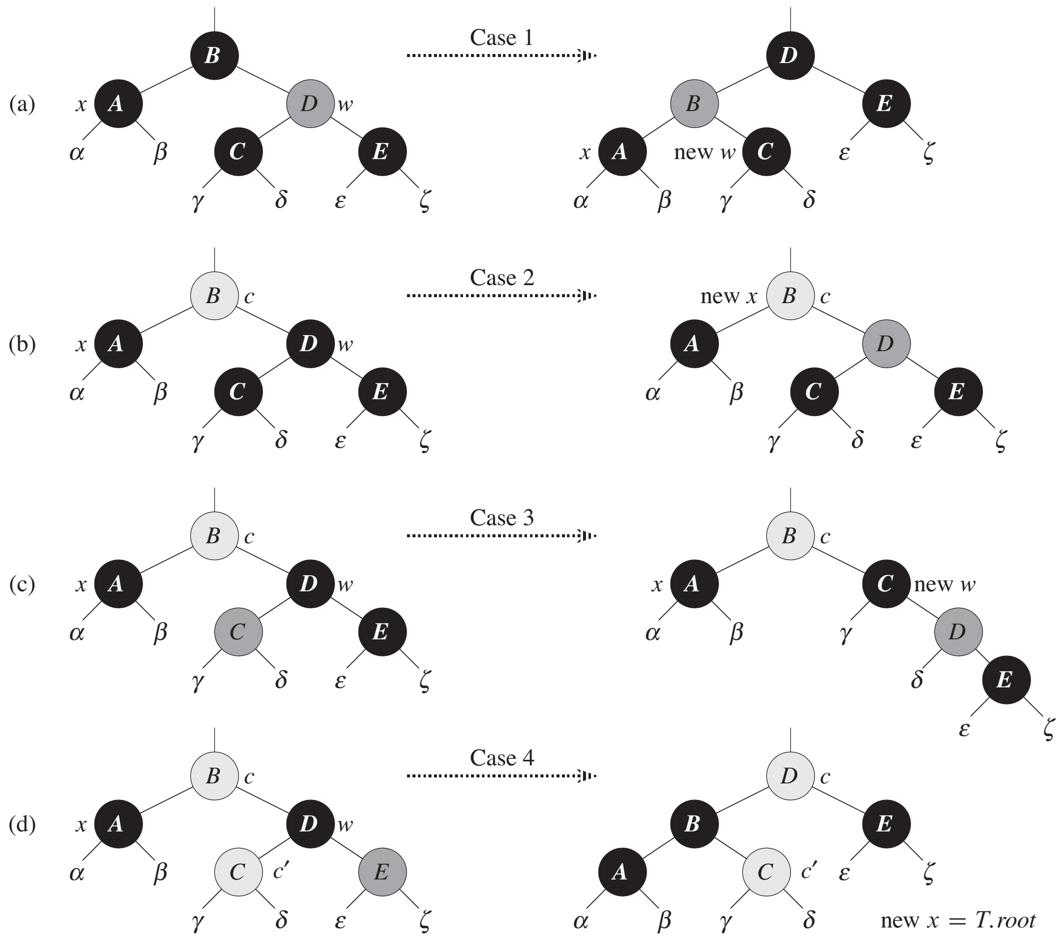


Figure 13.7 The cases in the **while** loop of the procedure RB-DELETE-FIXUP. Darkened nodes have *color* attributes BLACK, heavily shaded nodes have *color* attributes RED, and lightly shaded nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. **(b)** In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its *color* attribute is RED. **(c)** Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. **(d)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.

Exercises

13.4-1
Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

13.4-2
Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP(T, x).

13.4-3
In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order $8, 12, 19, 31, 38, 41$.

13.4-4
In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel $T.nil$?

13.4-5
In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees $\alpha, \beta, \ldots, \zeta$, and verify that each count remains the same after the transformation. When a node has a *color* attribute c or c', use the notation $\text{count}(c)$ or $\text{count}(c')$ symbolically in your count.

13.4-6
Professors Skelton and Baron are concerned that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If the professors are correct, then lines 5–6 are wrong. Show that $x.p$ must be black at the start of case 1, so that the professors have nothing to worry about.

13.4-7
Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

Problems

13-1 Persistent dynamic sets

During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. We call such a set ***persistent***. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume much space. Sometimes, we can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which we implement using binary search trees as shown in Figure 13.8(a). We maintain a separate root for every version of the set. In order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. We thus copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes *key*, *left*, and *right* but no parent. (See also Exercise 13.3-6.)

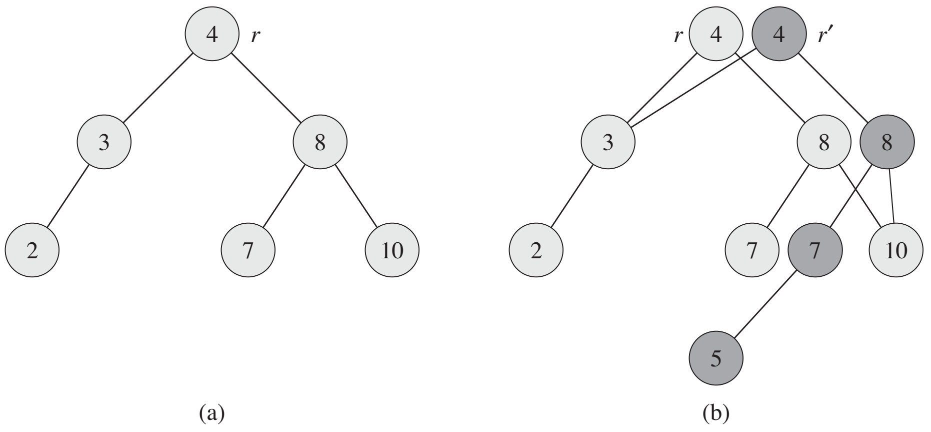


Figure 13.8 **(a)** A binary search tree with keys $2, 3, 4, 7, 8, 10$. **(b)** The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Heavily shaded nodes are added when key 5 is inserted.

a. For a general persistent binary search tree, identify the nodes that we need to change to insert a key k or delete a node y.

b. Write a procedure PERSISTENT-TREE-INSERT that, given a persistent tree T and a key k to insert, returns a new persistent tree T' that is the result of inserting k into T.

c. If the height of the persistent binary search tree T is h, what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of new nodes allocated.)

d. Suppose that we had included the parent attribute in each node. In this case, PERSISTENT-TREE-INSERT would need to perform additional copying. Prove that PERSISTENT-TREE-INSERT would then require $\Omega(n)$ time and space, where n is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion.

13-2 Join operation on red-black trees

The ***join*** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \le x.key \le x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Given a red-black tree T, let us store its black-height as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

We wish to implement the operation RB-JOIN(T_1, x, T_2), which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

b. Assume that $T_1.bh \ge T_2.bh$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.

c. Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

d. What color should we make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \le T_2.bh$.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.

13-3 AVL trees

An ***AVL tree*** is a binary search tree that is ***height balanced***: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x. As for any other binary search tree T, we assume that $T.root$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (*Hint:* Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.)

b. To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \le 2$, and alters the subtree rooted at x to be height balanced. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(x, z) that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = \text{NIL}$ and $z.right = \text{NIL}$; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T, we call AVL-INSERT$(T.root, z)$.

d. Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

13-4 Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. As we saw in Section 12.4, however, randomly built binary search trees tend to be balanced. Therefore, one strategy that, on average, builds a balanced tree for a fixed set of items would be to randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?

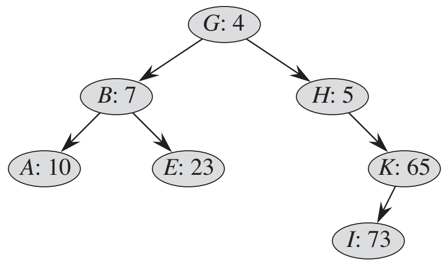


Figure 13.9 A treap. Each node x is labeled with $x.key : x.priority$. For example, the root has key G and priority 4.

We will examine a data structure that answers this question in the affirmative. A ***treap*** is a binary search tree with a modified way of ordering the nodes. Figure 13.9 shows an example. As usual, each node x in the tree has a key value $x.key$. In addition, we assign $x.priority$, which is a random number chosen independently for each node. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that the keys obey the binary-search-tree property and the priorities obey the min-heap order property:

- If v is a left child of u, then $v.key < u.key$.
- If v is a right child of u, then $v.key > u.key$.
- If v is a child of u, then $v.priority > u.priority$.

(This combination of properties is why the tree is called a "treap": it has features of both a binary search tree and a heap.)

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, with associated keys, into a treap. Then the resulting treap is the tree that would have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities, i.e., $x_i.priority < x_j.priority$ means that we had inserted x_i before x_j.

a. Show that given a set of nodes $x_1, x_2, \ldots, x_n$, with associated keys and priorities, all distinct, the treap associated with these nodes is unique.

b. Show that the expected height of a treap is $\Theta(\lg n)$, and hence the expected time to search for a value in the treap is $\Theta(\lg n)$.

Let us see how to insert a new node into an existing treap. The first thing we do is assign to the new node a random priority. Then we call the insertion algorithm, which we call TREAP-INSERT, whose operation is illustrated in Figure 13.10.

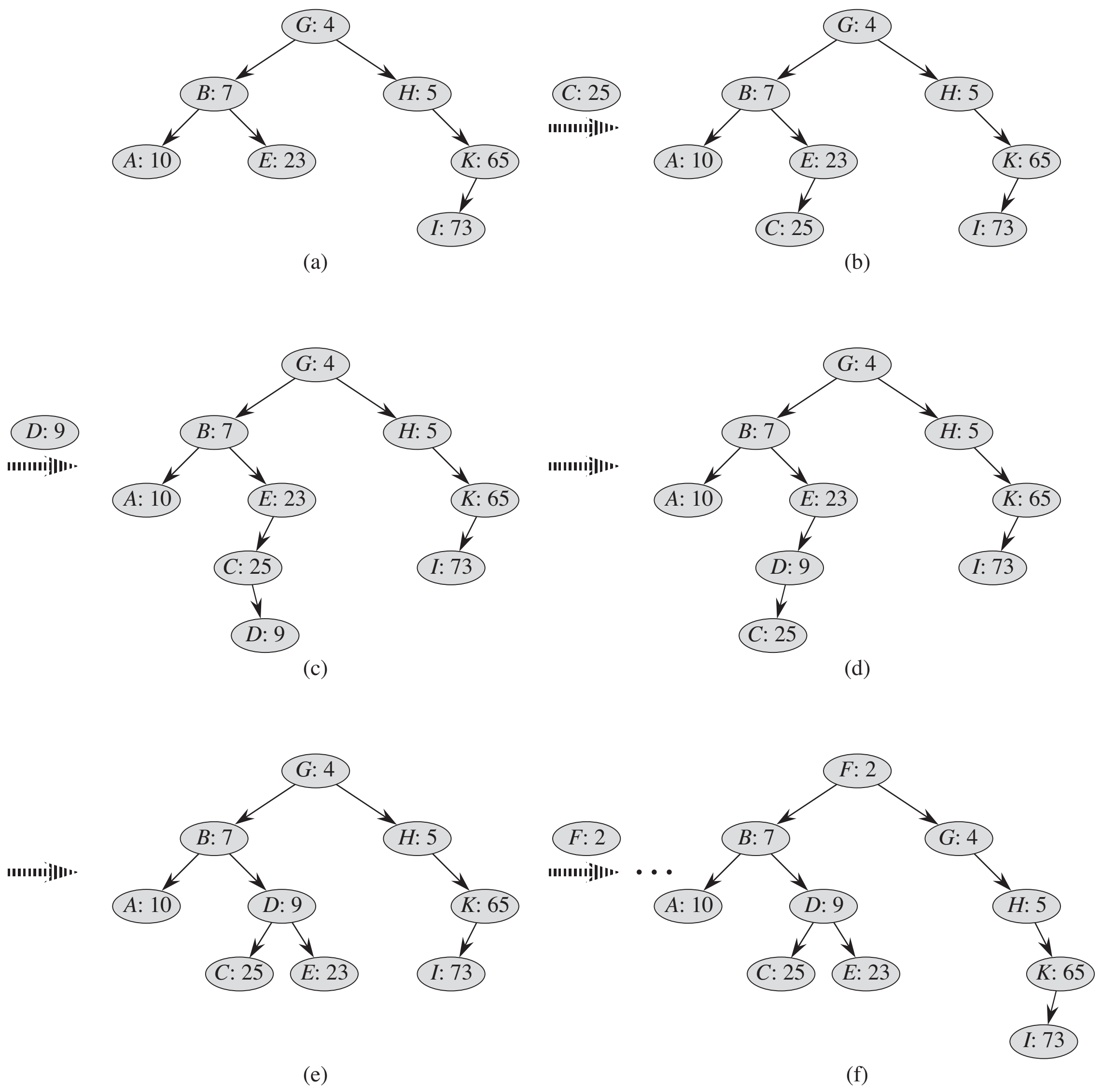


Figure 13.10 The operation of TREAP-INSERT. **(a)** The original treap, prior to insertion. **(b)** The treap after inserting a node with key C and priority 25. **(c)–(d)** Intermediate stages when inserting a node with key D and priority 9. **(e)** The treap after the insertion of parts (c) and (d) is done. **(f)** The treap after inserting a node with key F and priority 2.

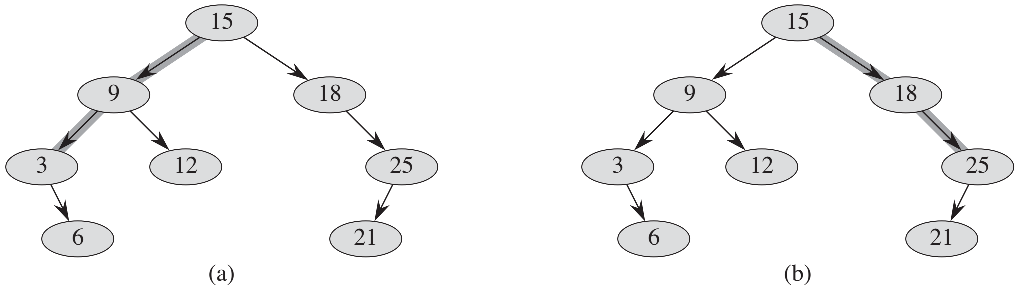


Figure 13.11 Spines of a binary search tree. The left spine is shaded in **(a)**, and the right spine is shaded in **(b)**.

c. Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (*Hint:* Execute the usual binary-search-tree insertion procedure and then perform rotations to restore the min-heap order property.)

d. Show that the expected running time of TREAP-INSERT is $\Theta(\lg n)$.

TREAP-INSERT performs a search and then a sequence of rotations. Although these two operations have the same expected running time, they have different costs in practice. A search reads information from the treap without modifying it. In contrast, a rotation changes parent and child pointers within the treap. On most computers, read operations are much faster than write operations. Thus we would like TREAP-INSERT to perform few rotations. We will show that the expected number of rotations performed is bounded by a constant.

In order to do so, we will need some definitions, which Figure 13.11 depicts. The ***left spine*** of a binary search tree T is the simple path from the root to the node with the smallest key. In other words, the left spine is the simple path from the root that consists of only left edges. Symmetrically, the ***right spine*** of T is the simple path from the root consisting of only right edges. The ***length*** of a spine is the number of nodes it contains.

e. Consider the treap T immediately after TREAP-INSERT has inserted node x. Let C be the length of the right spine of the left subtree of x. Let D be the length of the left spine of the right subtree of x. Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.

We will now calculate the expected values of C and D. Without loss of generality, we assume that the keys are $1, 2, \ldots, n$, since we are comparing them only to one another.

For nodes x and y in treap T, where $y \neq x$, let $k = x.key$ and $i = y.key$. We define indicator random variables

$$X_{ik} = \mathrm{I}\{y \text{ is in the right spine of the left subtree of } x\} .$$

f. Show that $X_{ik} = 1$ if and only if $y.priority > x.priority$, $y.key < x.key$, and, for every z such that $y.key < z.key < x.key$, we have $y.priority < z.priority$.

g. Show that

$$\begin{aligned} \Pr\{X_{ik} = 1\} &= \frac{(k-i-1)!}{(k-i+1)!} \\ &= \frac{1}{(k-i+1)(k-i)} . \end{aligned}$$

h. Show that

$$\begin{aligned} \mathrm{E}[C] &= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\ &= 1 - \frac{1}{k} . \end{aligned}$$

i. Use a symmetry argument to show that

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1} .$$

j. Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called "AVL trees" in 1962, described in Problem 13-3. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [35], called "B-trees."

Red-black trees were invented by Bayer [34] under the name "symmetric binary B-trees." Guibas and Sedgewick [155] studied their properties at length and introduced the red/black color convention. Andersson [15] gives a simpler-to-code variant of red-black trees. Weiss [351] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children may never be red.

Treaps, the subject of Problem 13-4, were proposed by Seidel and Aragon [309]. They are the default implementation of a dictionary in LEDA [253], which is a well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-balanced trees [264], k-neighbor trees [245], and scapegoat trees [127]. Perhaps the most intriguing are the "splay trees" introduced by Sleator and Tarjan [320], which are "self-adjusting." (See Tarjan [330] for a good description of splay trees.) Splay trees maintain balance without any explicit balance condition such as color. Instead, "splay operations" (which involve rotations) are performed within the tree every time an access is made. The amortized cost (see Chapter 17) of each operation on an n-node tree is $O(\lg n)$.

Skip lists [286] provide an alternative to balanced binary trees. A skip list is a linked list that is augmented with a number of additional pointers. Each dictionary operation runs in expected time $O(\lg n)$ on a skip list of n items.

Dynamic Programming

15 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ("Programming" in this context refers to a tabular method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

We typically apply dynamic programming to ***optimization problems***. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. Section 15.1 examines the problem of cutting a rod into rods of smaller length in way that maximizes their total value. Section 15.2 asks how we can multiply a chain of matrices while performing the fewest total scalar multiplications. Given these examples of dynamic programming, Section 15.3 discusses two key characteristics that a problem must have for dynamic programming to be a viable solution technique. Section 15.4 then shows how to find the longest common subsequence of two sequences via dynamic programming. Finally, Section 15.5 uses dynamic programming to construct binary search trees that are optimal, given a known distribution of keys to be looked up.

15.1 Rod cutting

Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

We assume that we know, for $i = 1, 2, \ldots$, the price p_i in dollars that Serling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches. Figure 15.1 gives a sample price table.

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Consider the case when $n = 4$. Figure 15.2 shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end,

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

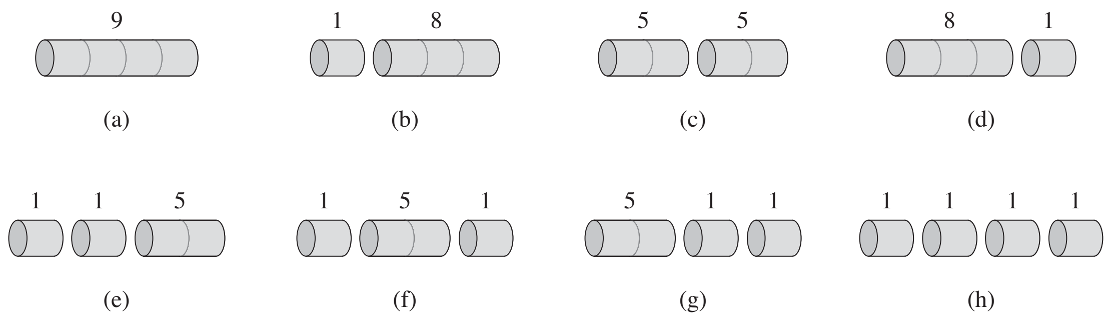


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

for $i = 1, 2, \ldots, n-1$.[1] We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3. If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

of the rod into pieces of lengths i_1, i_2, ..., i_k provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} \; .$$

For our sample problem, we can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

[1]If we required the pieces to be cut in order of nondecreasing size, there would be fewer ways to consider. For $n = 4$, we would consider only 5 such ways: parts (a), (b), (c), (e), and (h) in Figure 15.2. The number of ways is called the ***partition function***; it is approximately equal to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$. This quantity is less than 2^{n-1}, but still much greater than any polynomial in n. We shall not pursue this line of inquiry further, however.

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)}\ , \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)}\ , \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)}\ , \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2\ , \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3\ , \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)}\ , \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3\ , \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6\ , \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6\ , \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)}\ .
\end{aligned}
$$

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)\ . \tag{15.1}$$

The first argument, p_n, corresponds to making no cuts at all and selling the rod of length n as is. The other $n - 1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n - i$, for each $i = 1, 2, \ldots, n - 1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces. Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue. We also have the option of picking no i at all if we can obtain more revenue by selling the rod uncut.

Note that to solve the original problem of size n, we solve smaller problems of the same type, but of smaller sizes. Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces. We say that the rod-cutting problem exhibits ***optimal substructure***: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, we view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided. We may view every decomposition of a length-n rod in this way: as a first piece followed by some decomposition of the remainder. When doing so, we can couch the solution with no cuts at all as saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler version of equation (15.1):

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})\ . \tag{15.2}$$

In this formulation, an optimal solution embodies the solution to only *one* related subproblem—the remainder—rather than two.

Recursive top-down implementation

The following procedure implements the computation implicit in equation (15.2) in a straightforward, top-down, recursive manner.

```
CUT-ROD$(p, n)$
1  if $n == 0$
2      return 0
3  $q = -\infty$
4  for $i = 1$ to $n$
5      $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
6  return $q$
```

Procedure CUT-ROD takes as input an array $p[1\,..\,n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes $q = \max_{1 \le i \le n}(p_i + \text{CUT-ROD}(p, n - i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using equation (15.2).

If you were to code up CUT-ROD in your favorite programming language and run it on your computer, you would find that once the input size becomes moderately large, your program would take a long time to run. For $n = 40$, you would find that your program takes at least several minutes, and most likely more than an hour. In fact, you would find that each time you increase n by 1, your program's running time would approximately double.

Why is CUT-ROD so inefficient? The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values; it solves the same subproblems repeatedly. Figure 15.3 illustrates what happens for $n = 4$: CUT-ROD(p, n) calls CUT-ROD$(p, n - i)$ for $i = 1, 2, \ldots, n$. Equivalently, CUT-ROD(p, n) calls CUT-ROD(p, j) for each $j = 0, 1, \ldots, n - 1$. When this process unfolds recursively, the amount of work done, as a function of n, grows explosively.

To analyze the running time of CUT-ROD, let $T(n)$ denote the total number of calls made to CUT-ROD when called with its second parameter equal to n. This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree. The count includes the initial call at its root. Thus, $T(0) = 1$ and

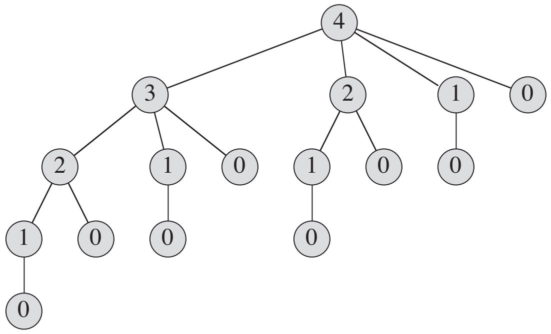


Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) . \tag{15.3}$$

The initial 1 is for the call at the root, and the term $T(j)$ counts the number of calls (including recursive calls) due to the call CUT-ROD$(p, n - i)$, where $j = n - i$. As Exercise 15.1-1 asks you to show,

$$T(n) = 2^n , \tag{15.4}$$

and so the running time of CUT-ROD is exponential in n.

In retrospect, this exponential running time is not so surprising. CUT-ROD explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n. The tree of recursive calls has 2^{n-1} leaves, one for each possible way of cutting up the rod. The labels on the simple path from the root to a leaf give the sizes of each remaining right-hand piece before making each cut. That is, the labels give the corresponding cut points, measured from the right-hand end of the rod.

Using dynamic programming for optimal rod cutting

We now show how to convert CUT-ROD into an efficient algorithm, using dynamic programming.

The dynamic-programming method works as follows. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only *once*, saving its solution. If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it. Dynamic programming thus uses additional memory to save computation time; it serves an example of a ***time-memory trade-off***. The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution. A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach. We shall illustrate both of them with our rod-cutting example.

The first approach is ***top-down with memoization***.[2] In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner. We say that the recursive procedure has been ***memoized***; it "remembers" what results it has computed previously.

The second approach is the ***bottom-up method***. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Here is the the pseudocode for the top-down CUT-ROD procedure, with memoization added:

```
MEMOIZED-CUT-ROD($p, n$)
1  let $r[0..n]$ be a new array
2  for $i = 0$ to $n$
3      $r[i] = -\infty$
4  return MEMOIZED-CUT-ROD-AUX($p, n, r$)
```

[2]This is not a misspelling. The word really is *memoization*, not *memorization*. *Memoization* comes from *memo*, since the technique consists of recording a value so that we can look it up later.

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
1  if $r[n] \geq 0$
2      return $r[n]$
3  if $n == 0$
4      $q = 0$
5  else $q = -\infty$
6      for $i = 1$ to $n$
7          $q = \max(q, p[i] + $MEMOIZED-CUT-ROD-AUX$(p, n-i, r))$
8  $r[n] = q$
9  return $q$
```

Here, the main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0..n]$ with the value $-\infty$, a convenient choice with which to denote "unknown." (Known revenue values are always nonnegative.) It then calls its helper routine, MEMOIZED-CUT-ROD-AUX.

The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of our previous procedure, CUT-ROD. It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $r[n]$, and line 9 returns it.

The bottom-up version is even simpler:

```
BOTTOM-UP-CUT-ROD$(p, n)$
1  let $r[0..n]$ be a new array
2  $r[0] = 0$
3  for $j = 1$ to $n$
4      $q = -\infty$
5      for $i = 1$ to $j$
6          $q = \max(q, p[i] + r[j-i])$
7      $r[j] = q$
8  return $r[n]$
```

For the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD uses the natural ordering of the subproblems: a problem of size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Line 1 of procedure BOTTOM-UP-CUT-ROD creates a new array $r[0..n]$ in which to save the results of the subproblems, and line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue. Lines 3–6 solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now

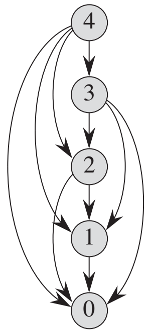


Figure 15.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x. This graph is a reduced version of the tree of Figure 15.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

directly references array entry $r[j - i]$ instead of making a recursive call to solve the subproblem of size $j - i$. Line 7 saves in $r[j]$ the solution to the subproblem of size j. Finally, line 8 returns $r[n]$, which equals the optimal value r_n.

The bottom-up and top-down versions have the same asymptotic running time. The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure. The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series. The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is also $\Theta(n^2)$, although this running time may be a little harder to see. Because a recursive call to solve a previously solved subproblem returns immediately, MEMOIZED-CUT-ROD solves each subproblem just once. It solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop of lines 6–7 iterates n times. Thus, the total number of iterations of this **for** loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series, giving a total of $\Theta(n^2)$ iterations, just like the inner **for** loop of BOTTOM-UP-CUT-ROD. (We actually are using a form of aggregate analysis here. We shall see aggregate analysis in detail in Section 17.1.)

Subproblem graphs

When we think about a dynamic-programming problem, we should understand the set of subproblems involved and how subproblems depend on one another.

The ***subproblem graph*** for the problem embodies exactly this information. Figure 15.4 shows the subproblem graph for the rod-cutting problem with $n = 4$. It is a directed graph, containing one vertex for each distinct subproblem. The subproblem graph has a directed edge from the vertex for subproblem x to the vertex for subproblem y if determining an optimal solution for subproblem x involves directly considering an optimal solution for subproblem y. For example, the subproblem graph contains an edge from x to y if a top-down recursive procedure for solving x directly calls itself to solve y. We can think of the subproblem graph as a "reduced" or "collapsed" version of the recursion tree for the top-down recursive method, in which we coalesce all nodes for the same subproblem into a single vertex and direct all edges from parent to child.

The bottom-up method for dynamic programming considers the vertices of the subproblem graph in such an order that we solve the subproblems y adjacent to a given subproblem x before we solve subproblem x. (Recall from Section B.4 that the adjacency relation is not necessarily symmetric.) Using the terminology from Chapter 22, in a bottom-up dynamic-programming algorithm, we consider the vertices of the subproblem graph in an order that is a "reverse topological sort," or a "topological sort of the transpose" (see Section 22.4) of the subproblem graph. In other words, no subproblem is considered until all of the subproblems it depends upon have been solved. Similarly, using notions from the same chapter, we can view the top-down method (with memoization) for dynamic programming as a "depth-first search" of the subproblem graph (see Section 22.3).

The size of the subproblem graph $G = (V, E)$ can help us determine the running time of the dynamic programming algorithm. Since we solve each subproblem just once, the running time is the sum of the times needed to solve each subproblem. Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph. In this common case, the running time of dynamic programming is linear in the number of vertices and edges.

Reconstructing a solution

Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes. We can extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, we can readily print an optimal solution.

Here is an extended version of BOTTOM-UP-CUT-ROD that computes, for each rod size j, not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off:

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let $r[0..n]$ and $s[0..n]$ be new arrays
2  $r[0] = 0$
3  for $j = 1$ to $n$
4      $q = -\infty$
5      for $i = 1$ to $j$
6          if $q < p[i] + r[j-i]$
7              $q = p[i] + r[j-i]$
8              $s[j] = i$
9      $r[j] = q$
10 return $r$ and $s$
```

This procedure is similar to BOTTOM-UP-CUT-ROD, except that it creates the array s in line 1, and it updates $s[j]$ in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

The following procedure takes a price table p and a rod size n, and it calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1..n]$ of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n:

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  $(r, s)$ = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while $n > 0$
3      print $s[n]$
4      $n = n - s[n]$
```

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

A call to PRINT-CUT-ROD-SOLUTION$(p, 10)$ would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7 given earlier.

Exercises

15.1-1

Show that equation (15.4) follows from equation (15.3) and the initial condition $T(0) = 1$.

15.1-2

Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

15.1-3

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

15.1-4

Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution, too.

15.1-5

The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

15.2 Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \cdots A_n . \qquad (15.5)$$

We can evaluate the expression (15.5) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then we can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode, which generalizes the SQUARE-MATRIX-MULTIPLY procedure from Section 4.2. The attributes *rows* and *columns* are the numbers of rows and columns in a matrix.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              $c_{ij} = 0$
7              for k = 1 to A.columns
8                  $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
9      return C
```

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr. In what follows, we shall express costs in terms of the number of scalar multiplications.

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. If we multiply according to the parenthesization $((A_1A_2)A_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product A_1A_2, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization $(A_1(A_2A_3))$, we perform $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product A_2A_3, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the ***matrix-chain multiplication problem*** as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension

$p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \, , \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \, . \end{cases} \tag{15.6}$$

Problem 12-4 asked you to show that the solution to a similar recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n/n^{3/2})$. A simpler exercise (see Exercise 15.2-3) is to show that the solution to the recurrence (15.6) is $\Omega(2^n)$. The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Applying dynamic programming

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain. In so doing, we shall follow the four-step sequence that we stated at the beginning of this chapter:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.

4. Construct an optimal solution from computed information.

We shall go through these steps in order, demonstrating clearly how we apply each step to the problem.

Step 1: The structure of an optimal parenthesization

For our first step in the dynamic-programming paradigm, we find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. In the matrix-chain multiplication problem, we can perform this step as follows. For convenience, let us adopt the notation $A_{i..j}$, where $i \le j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Now we use our optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. We have seen that any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product, and that any optimal solution contains within it optimal solutions to subproblem instances. Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions. We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.

Step 2: A recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that to optimally parenthesize, we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i, j]$ equals the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \ .$$

This recursive equation assumes that we know the value of k, which we do not. There are only $j - i$ possible values for k, however, namely $k = i, i+1, \ldots, j-1$. Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \ . \end{cases} \qquad (15.7)$$

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$.

Step 3: Computing the optimal costs

At this point, we could easily write a recursive algorithm based on recurrence (15.7) to compute the minimum cost $m[1, n]$ for multiplying $A_1 A_2 \cdots A_n$. As we saw for the rod-cutting problem, and as we shall see in Section 15.3, this recursive algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.

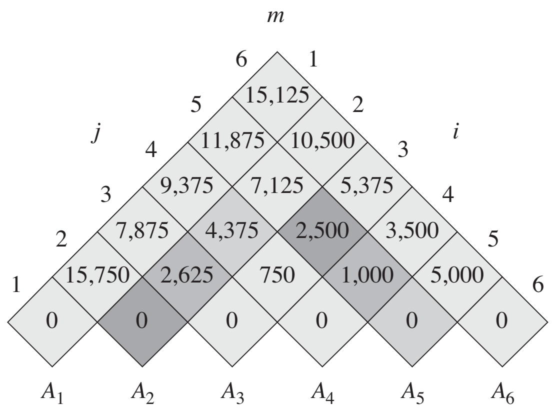
Observe that we have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence (15.7) recursively, we compute the optimal cost by using a tabular, bottom-up approach. (We present the corresponding top-down approach using memoization in Section 15.3.)

We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears below. This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.\mathit{length} = n + 1$. The procedure uses an auxiliary table $m[1 \mathinner{..} n, 1 \mathinner{..} n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1 \mathinner{..} n-1, 2 \mathinner{..} n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$. We shall use the table s to construct an optimal solution.

In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing $m[i, j]$. Equation (15.7) shows that the cost $m[i, j]$ of computing a matrix-chain product of $j - i + 1$ matrices depends only on the costs of computing matrix-chain products of fewer than $j - i + 1$ matrices. That is, for $k = i, i+1, \ldots, j-1$, the matrix $A_{i..k}$ is a product of $k - i + 1 < j - i + 1$ matrices and the matrix $A_{k+1..j}$ is a product of $j - k < j - i + 1$ matrices. Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. For the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, we consider the subproblem size to be the length $j - i + 1$ of the chain.

```
MATRIX-CHAIN-ORDER($p$)
1   $n = p.length - 1$
2   let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
3   **for** $i = 1$ **to** $n$
4       $m[i, i] = 0$
5   **for** $l = 2$ **to** $n$            // $l$ is the chain length
6       **for** $i = 1$ **to** $n - l + 1$
7           $j = i + l - 1$
8           $m[i, j] = \infty$
9           **for** $k = i$ **to** $j - 1$
10              $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
11              **if** $q < m[i, j]$
12                  $m[i, j] = q$
13                  $s[i, j] = k$
14  **return** $m$ and $s$
```



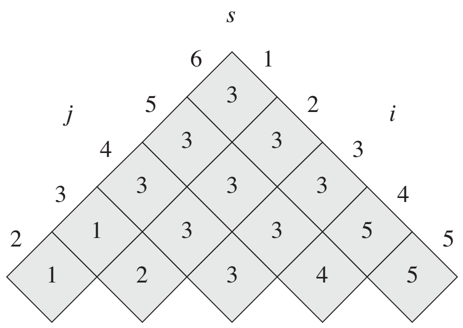


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125 .$$

The algorithm first computes $m[i,i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1) in lines 3–4. It then uses recurrence (15.7) to compute $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines 5–13. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i,j]$ cost computed in lines 10–13 depends only on table entries $m[i,k]$ and $m[k+1, j]$ already computed.

Figure 15.5 illustrates this procedure on a chain of $n = 6$ matrices. Since we have defined $m[i,j]$ only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, we can find the minimum cost $m[i,j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices at the intersection of lines running northeast from A_i and northwest from A_j. Each horizontal row in the table contains the entries for matrix chains of the same length. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i, j]$ using the products $p_{i-1} p_k p_j$ for $k = i, i+1, \ldots, j-1$ and all entries southwest and southeast from $m[i, j]$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l, i, and k) takes on at most $n-1$ values. Exercise 15.2-5 asks you to show that the running time of this algorithm is in fact also $\Omega(n^3)$. The algorithm requires $\Theta(n^2)$ space to store the m and s tables. Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1 \mathinner{.\,.} n-1, 2 \mathinner{.\,.} n]$ gives us the information we need to do so. Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$. We can determine the earlier matrix multiplications recursively, since $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1, n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$. The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

Exercises

15.2-1
Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

15.2-2
Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call would be MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.)

15.2-3
Use the substitution method to show that the solution to the recurrence (15.6) is $\Omega(2^n)$.

15.2-4
Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

15.2-5
Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total number of references for the entire table is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3} .$$

(*Hint:* You may find equation (A.3) useful.)

15.2-6
Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

15.3 Elements of dynamic programming

Although we have just worked through two examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should we look for a dynamic-programming solution to a problem? In this section, we examine the two key ingredients that an optimization problem must have in order for dynamic programming to apply: optimal substructure and overlapping subproblems. We also revisit and discuss more fully how memoization might help us take advantage of the overlapping-subproblems property in a top-down recursive approach.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply. (As Chapter 16 discusses, it also might mean that a greedy strategy applies, however.) In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

We discovered optimal substructure in both of the problems we have examined in this chapter so far. In Section 15.1, we observed that the optimal way of cutting up a rod of length n (if we make any cuts at all) involves optimally cutting up the two pieces resulting from the first cut. In Section 15.2, we observed that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.

2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.

3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.

4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by "cutting out" the nonoptimal solution to each subproblem and "pasting in" the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If an optimal

solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

To characterize the space of subproblems, a good rule of thumb says to try to keep the space as simple as possible and then expand it as necessary. For example, the space of subproblems that we considered for the rod-cutting problem contained the problems of optimally cutting up a rod of length i for each size i. This subproblem space worked well, and we had no need to try a more general space of subproblems.

Conversely, suppose that we had tried to constrain our subproblem space for matrix-chain multiplication to matrix products of the form $A_1 A_2 \cdots A_j$. As before, an optimal parenthesization must split this product between A_k and A_{k+1} for some $1 \le k < j$. Unless we could guarantee that k always equals $j-1$, we would find that we had subproblems of the form $A_1 A_2 \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$, and that the latter subproblem is not of the form $A_1 A_2 \cdots A_j$. For this problem, we needed to allow our subproblems to vary at "both ends," that is, to allow both i and j to vary in the subproblem $A_i A_{i+1} \cdots A_j$.

Optimal substructure varies across problem domains in two ways:

1. how many subproblems an optimal solution to the original problem uses, and
2. how many choices we have in determining which subproblem(s) to use in an optimal solution.

In the rod-cutting problem, an optimal solution for cutting up a rod of size n uses just one subproblem (of size $n-i$), but we must consider n choices for i in order to determine which one yields an optimal solution. Matrix-chain multiplication for the subchain $A_i A_{i+1} \cdots A_j$ serves as an example with two subproblems and $j-i$ choices. For a given matrix A_k at which we split the product, we have two subproblems—parenthesizing $A_i A_{i+1} \cdots A_k$ and parenthesizing $A_{k+1} A_{k+2} \cdots A_j$—and we must solve *both* of them optimally. Once we determine the optimal solutions to subproblems, we choose from among $j-i$ candidates for the index k.

Informally, the running time of a dynamic-programming algorithm depends on the product of two factors: the number of subproblems overall and how many choices we look at for each subproblem. In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $O(n^2)$ running time. Matrix-chain multiplication had $\Theta(n^2)$ subproblems overall, and in each we had at most $n-1$ choices, giving an $O(n^3)$ running time (actually, a $\Theta(n^3)$ running time, by Exercise 15.2-5).

Usually, the subproblem graph gives an alternative way to perform the same analysis. Each vertex corresponds to a subproblem, and the choices for a subproblem are the edges incident to that subproblem. Recall that in rod cutting, the subproblem graph had n vertices and at most n edges per vertex, yielding an $O(n^2)$ running time. For matrix-chain multiplication, if we were to draw the subproblem graph, it would have $\Theta(n^2)$ vertices and each vertex would have degree at most $n-1$, giving a total of $O(n^3)$ vertices and edges.

Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem. Finding an optimal solution to the problem entails making a choice among subproblems as to which we will use in solving the problem. The cost of the problem solution is usually the subproblem costs plus a cost that is directly attributable to the choice itself. In rod cutting, for example, first we solved the subproblems of determining optimal ways to cut up rods of length i for $i = 0, 1, \ldots, n-1$, and then we determined which such subproblem yielded an optimal solution for a rod of length n, using equation (15.2). The cost attributable to the choice itself is the term p_i in equation (15.2). In matrix-chain multiplication, we determined optimal parenthesizations of subchains of $A_i A_{i+1} \cdots A_j$, and then we chose the matrix A_k at which to split the product. The cost attributable to the choice itself is the term $p_{i-1} p_k p_j$.

In Chapter 16, we shall examine "greedy algorithms," which have many similarities to dynamic programming. In particular, problems to which greedy algorithms apply have optimal substructure. One major difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to subproblems and then making an informed choice, greedy algorithms first make a "greedy" choice—the choice that looks best at the time—and then solve a resulting subproblem, without bothering to solve all possible related smaller subproblems. Surprisingly, in some cases this strategy works!

Subtleties

You should be careful not to assume that optimal substructure applies when it does not. Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.

Unweighted shortest path:[3] Find a path from u to v consisting of the fewest edges. Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.

[3]We use the term "unweighted" to distinguish this problem from that of finding shortest paths with weighted edges, which we shall see in Chapters 24 and 25. We can use the breadth-first search technique of Chapter 22 to solve the unweighted problem.

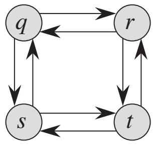


Figure 15.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. We need to include the requirement of simplicity because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.

The unweighted shortest-path problem exhibits optimal substructure, as follows. Suppose that $u \neq v$, so that the problem is nontrivial. Then, any path p from u to v must contain an intermediate vertex, say w. (Note that w may be u or v.) Thus, we can decompose the path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$. Clearly, the number of edges in p equals the number of edges in p_1 plus the number of edges in p_2. We claim that if p is an optimal (i.e., shortest) path from u to v, then p_1 must be a shortest path from u to w. Why? We use a "cut-and-paste" argument: if there were another path, say p'_1, from u to w with fewer edges than p_1, then we could cut out p_1 and paste in p'_1 to produce a path $u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$ with fewer edges than p, thus contradicting p's optimality. Symmetrically, p_2 must be a shortest path from w to v. Thus, we can find a shortest path from u to v by considering all intermediate vertices w, finding a shortest path from u to w and a shortest path from w to v, and choosing an intermediate vertex w that yields the overall shortest path. In Section 25.2, we use a variant of this observation of optimal substructure to find a shortest path between every pair of vertices on a weighted, directed graph.

You might be tempted to assume that the problem of finding an unweighted longest simple path exhibits optimal substructure as well. After all, if we decompose a longest simple path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, then mustn't p_1 be a longest simple path from u to w, and mustn't p_2 be a longest simple path from w to v? The answer is no! Figure 15.6 supplies an example. Consider the path $q \to r \to t$, which is a longest simple path from q to t. Is $q \to r$ a longest simple path from q to r? No, for the path $q \to s \to t \to r$ is a simple path that is longer. Is $r \to t$ a longest simple path from r to t? No again, for the path $r \to q \to s \to t$ is a simple path that is longer.

This example shows that for longest simple paths, not only does the problem lack optimal substructure, but we cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems. If we combine the longest simple paths $q \to s \to t \to r$ and $r \to q \to s \to t$, we get the path $q \to s \to t \to r \to q \to s \to t$, which is not simple. Indeed, the problem of finding an unweighted longest simple path does not appear to have any sort of optimal substructure. No efficient dynamic-programming algorithm for this problem has ever been found. In fact, this problem is NP-complete, which—as we shall see in Chapter 34—means that we are unlikely to find a way to solve it in polynomial time.

Why is the substructure of a longest simple path so different from that of a shortest path? Although a solution to a problem for both longest and shortest paths uses two subproblems, the subproblems in finding the longest simple path are not ***independent***, whereas for shortest paths they are. What do we mean by subproblems being independent? We mean that the solution to one subproblem does not affect the solution to another subproblem of the same problem. For the example of Figure 15.6, we have the problem of finding a longest simple path from q to t with two subproblems: finding longest simple paths from q to r and from r to t. For the first of these subproblems, we choose the path $q \to s \to t \to r$, and so we have also used the vertices s and t. We can no longer use these vertices in the second subproblem, since the combination of the two solutions to subproblems would yield a path that is not simple. If we cannot use vertex t in the second problem, then we cannot solve it at all, since t is required to be on the path that we find, and it is not the vertex at which we are "splicing" together the subproblem solutions (that vertex being r). Because we use vertices s and t in one subproblem solution, we cannot use them in the other subproblem solution. We must use at least one of them to solve the other subproblem, however, and we must use both of them to solve it optimally. Thus, we say that these subproblems are not independent. Looked at another way, using resources in solving one subproblem (those resources being vertices) renders them unavailable for the other subproblem.

Why, then, are the subproblems independent for finding a shortest path? The answer is that by nature, the subproblems do not share resources. We claim that if a vertex w is on a shortest path p from u to v, then we can splice together *any* shortest path $u \overset{p_1}{\leadsto} w$ and *any* shortest path $w \overset{p_2}{\leadsto} v$ to produce a shortest path from u to v. We are assured that, other than w, no vertex can appear in both paths p_1 and p_2. Why? Suppose that some vertex $x \neq w$ appears in both p_1 and p_2, so that we can decompose p_1 as $u \overset{p_{ux}}{\leadsto} x \leadsto w$ and p_2 as $w \leadsto x \overset{p_{xv}}{\leadsto} v$. By the optimal substructure of this problem, path p has as many edges as p_1 and p_2 together; let's say that p has e edges. Now let us construct a path $p' = u \overset{p_{ux}}{\leadsto} x \overset{p_{xv}}{\leadsto} v$ from u to v. Because we have excised the paths from x to w and from w to x, each of which contains at least one edge, path p' contains at most $e - 2$ edges, which contradicts the assumption that p is a shortest path. Thus, we are assured that the subproblems for the shortest-path problem are independent.

Both problems examined in Sections 15.1 and 15.2 have independent subproblems. In matrix-chain multiplication, the subproblems are multiplying subchains $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. These subchains are disjoint, so that no matrix could possibly be included in both of them. In rod cutting, to determine the best way to cut up a rod of length n, we look at the best ways of cutting up rods of length i for $i = 0, 1, \ldots, n-1$. Because an optimal solution to the length-n problem includes just one of these subproblem solutions (after we have cut off the first piece), independence of subproblems is not an issue.

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping subproblems***.[4] In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

In Section 15.1, we briefly examined how a recursive solution to rod cutting makes exponentially many calls to find solutions of smaller subproblems. Our dynamic-programming solution takes an exponential-time recursive algorithm down to quadratic time.

To illustrate the overlapping-subproblems property in greater detail, let us reexamine the matrix-chain multiplication problem. Referring back to Figure 15.5, observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows. For example, it references entry $m[3,4]$ four times: during the computations of $m[2,4]$, $m[1,4]$,

[4]It may seem strange that dynamic programming relies on subproblems being both independent and overlapping. Although these requirements may sound contradictory, they describe two different notions, rather than two points on the same axis. Two subproblems of the same problem are independent if they do not share resources. Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems.

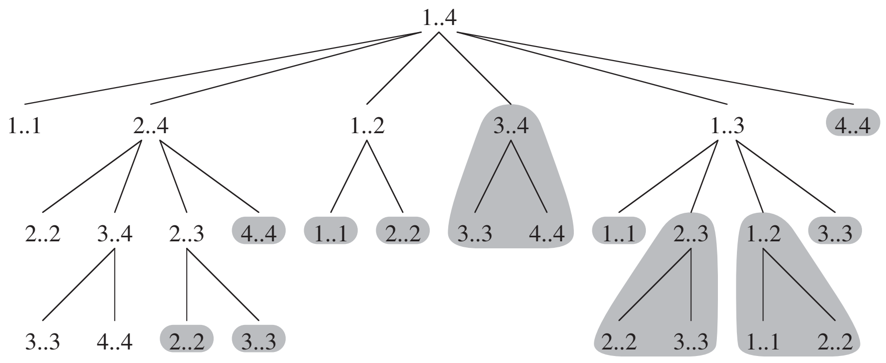


Figure 15.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

$m[3, 5]$, and $m[3, 6]$. If we were to recompute $m[3, 4]$ each time, rather than just looking it up, the running time would increase dramatically. To see how, consider the following (inefficient) recursive procedure that determines $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix-chain product $A_{i..j} = A_i A_{i+1} \cdots A_j$. The procedure is based directly on the recurrence (15.7).

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if $i == j$
2      return 0
3  $m[i, j] = \infty$
4  for $k = i$ to $j - 1$
5      $q$ = RECURSIVE-MATRIX-CHAIN$(p, i, k)$
             + RECURSIVE-MATRIX-CHAIN$(p, k + 1, j)$
             + $p_{i-1} p_k p_j$
6      if $q < m[i, j]$
7          $m[i, j] = q$
8  return $m[i, j]$
```

Figure 15.7 shows the recursion tree produced by the call RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node is labeled by the values of the parameters i and j. Observe that some pairs of values occur many times.

In fact, we can show that the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n. Let $T(n)$ denote the time taken by RECURSIVE-MATRIX-CHAIN to compute an optimal parenthesization of a chain of n matrices. Because the execution of lines 1–2 and of lines 6–7 each take at least unit time, as does the multiplication in line 5, inspection of the procedure yields the recurrence

$$\begin{aligned} T(1) &\geq 1 , \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1 . \end{aligned}$$

Noting that for $i = 1, 2, \ldots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n . \tag{15.8}$$

We shall prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we shall show that $T(n) \geq 2^{n-1}$ for all $n \geq 1$. The basis is easy, since $T(1) \geq 1 = 2^0$. Inductively, for $n \geq 2$ we have

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \quad \text{(by equation (A.5))} \\ &= 2^n - 2 + n \\ &\geq 2^{n-1} , \end{aligned}$$

which completes the proof. Thus, the total amount of work performed by the call RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ is at least exponential in n.

Compare this top-down, recursive algorithm (without memoization) with the bottom-up dynamic-programming algorithm. The latter is more efficient because it takes advantage of the overlapping-subproblems property. Matrix-chain multiplication has only $\Theta(n^2)$ distinct subproblems, and the dynamic-programming algorithm solves each exactly once. The recursive algorithm, on the other hand, must again solve each subproblem every time it reappears in the recursion tree. Whenever a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly, and the total number of distinct subproblems is small, dynamic programming can improve efficiency, sometimes dramatically.

Reconstructing an optimal solution

As a practical matter, we often store which choice we made in each subproblem in a table so that we do not have to reconstruct this information from the costs that we stored.

For matrix-chain multiplication, the table $s[i, j]$ saves us a significant amount of work when reconstructing an optimal solution. Suppose that we did not maintain the $s[i, j]$ table, having filled in only the table $m[i, j]$ containing optimal subproblem costs. We choose from among $j - i$ possibilities when we determine which subproblems to use in an optimal solution to parenthesizing $A_i A_{i+1} \cdots A_j$, and $j - i$ is not a constant. Therefore, it would take $\Theta(j - i) = \omega(1)$ time to reconstruct which subproblems we chose for a solution to a given problem. By storing in $s[i, j]$ the index of the matrix at which we split the product $A_i A_{i+1} \cdots A_j$, we can reconstruct each choice in $O(1)$ time.

Memoization

As we saw for the rod-cutting problem, there is an alternative approach to dynamic programming that often offers the efficiency of the bottom-up dynamic-programming approach while maintaining a top-down strategy. The idea is to ***memoize*** the natural, but inefficient, recursive algorithm. As in the bottom-up approach, we maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.

A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table. Each subsequent time that we encounter this subproblem, we simply look up the value stored in the table and return it.[5]

Here is a memoized version of RECURSIVE-MATRIX-CHAIN. Note where it resembles the memoized top-down method for the rod-cutting problem.

[5]This approach presupposes that we know the set of all possible subproblem parameters and that we have established the relationship between table positions and subproblems. Another, more general, approach is to memoize by using hashing with the subproblem parameters as keys.

```
MEMOIZED-MATRIX-CHAIN($p$)
1  $n = p.length - 1$
2  let $m[1..n, 1..n]$ be a new table
3  for $i = 1$ to $n$
4      for $j = i$ to $n$
5          $m[i, j] = \infty$
6  return LOOKUP-CHAIN($m, p, 1, n$)
```

```
LOOKUP-CHAIN($m, p, i, j$)
1  if $m[i, j] < \infty$
2      return $m[i, j]$
3  if $i == j$
4      $m[i, j] = 0$
5  else for $k = i$ to $j - 1$
6          $q$ = LOOKUP-CHAIN($m, p, i, k$)
                + LOOKUP-CHAIN($m, p, k + 1, j$) + $p_{i-1} p_k p_j$
7          if $q < m[i, j]$
8              $m[i, j] = q$
9  return $m[i, j]$
```

The MEMOIZED-MATRIX-CHAIN procedure, like MATRIX-CHAIN-ORDER, maintains a table $m[1..n, 1..n]$ of computed values of $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$. Each table entry initially contains the value ∞ to indicate that the entry has yet to be filled in. Upon calling LOOKUP-CHAIN(m, p, i, j), if line 1 finds that $m[i, j] < \infty$, then the procedure simply returns the previously computed cost $m[i, j]$ in line 2. Otherwise, the cost is computed as in RECURSIVE-MATRIX-CHAIN, stored in $m[i, j]$, and returned. Thus, LOOKUP-CHAIN(m, p, i, j) always returns the value of $m[i, j]$, but it computes it only upon the first call of LOOKUP-CHAIN with these specific values of i and j.

Figure 15.7 illustrates how MEMOIZED-MATRIX-CHAIN saves time compared with RECURSIVE-MATRIX-CHAIN. Shaded subtrees represent values that it looks up rather than recomputes.

Like the bottom-up dynamic-programming algorithm MATRIX-CHAIN-ORDER, the procedure MEMOIZED-MATRIX-CHAIN runs in $O(n^3)$ time. Line 5 of MEMOIZED-MATRIX-CHAIN executes $\Theta(n^2)$ times. We can categorize the calls of LOOKUP-CHAIN into two types:

1. calls in which $m[i, j] = \infty$, so that lines 3–9 execute, and
2. calls in which $m[i, j] < \infty$, so that LOOKUP-CHAIN simply returns in line 2.

There are $\Theta(n^2)$ calls of the first type, one per table entry. All calls of the second type are made as recursive calls by calls of the first type. Whenever a given call of LOOKUP-CHAIN makes recursive calls, it makes $O(n)$ of them. Therefore, there are $O(n^3)$ calls of the second type in all. Each call of the second type takes $O(1)$ time, and each call of the first type takes $O(n)$ time plus the time spent in its recursive calls. The total time, therefore, is $O(n^3)$. Memoization thus turns an $\Omega(2^n)$-time algorithm into an $O(n^3)$-time algorithm.

In summary, we can solve the matrix-chain multiplication problem by either a top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-programming algorithm in $O(n^3)$ time. Both methods take advantage of the overlapping-subproblems property. There are only $\Theta(n^2)$ distinct subproblems in total, and either of these methods computes the solution to each subproblem only once. Without memoization, the natural recursive algorithm runs in exponential time, since solved subproblems are repeatedly solved.

In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table. Moreover, for some problems we can exploit the regular pattern of table accesses in the dynamic-programming algorithm to reduce time or space requirements even further. Alternatively, if some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Exercises

15.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

15.3-2

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

15.3-3

Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

15.3-4

As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) *before* solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

15.3-5

Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

15.3-6

Imagine that you wish to exchange one currency for another. You realize that instead of directly exchanging one currency for another, you might be better off making a series of trades through other currencies, winding up with the currency you want. Suppose that you can trade n different currencies, numbered $1, 2, \ldots, n$, where you start with currency 1 and wish to wind up with currency n. You are given, for each pair of currencies i and j, an exchange rate r_{ij}, meaning that if you start with d units of currency i, you can trade for dr_{ij} units of currency j. A sequence of trades may entail a commission, which depends on the number of trades you make. Let c_k be the commission that you are charged when you make k trades. Show that, if $c_k = 0$ for all $k = 1, 2, \ldots, n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure. Then show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

15.4 Longest common subsequence

Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called

bases, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. (See Appendix C for the definition of a string.) For example, the DNA of one organism may be $S_1 = \mathtt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$, and the DNA of another organism may be $S_2 = \mathtt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$. One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are. We can, and do, define similarity in many different ways. For example, we can say that two DNA strands are similar if one is a substring of the other. (Chapter 32 explores algorithms to solve this problem.) In our example, neither S_1 nor S_2 is a substring of the other. Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. (Problem 15-5 looks at this notion.) Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In our example, the longest strand S_3 is $\mathtt{GTCGTCGGAAGCCGGCCGAA}$.

We formalize this last notion of similarity as the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y. For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y. The sequence $\langle B, C, A \rangle$ is not a *longest* common subsequence (LCS) of X and Y, however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both X and Y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y, as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater.

In the ***longest-common-subsequence problem***, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y. This section shows how to efficiently solve the LCS problem using dynamic programming.

Step 1: Characterizing a longest common subsequence

In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X. Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

The LCS problem has an optimal-substructure property, however, as the following theorem shows. As we shall see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences. To be precise, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the ith ***prefix*** of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k - 1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2). ■

The way that Theorem 15.1 characterizes longest common subsequences tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property, as we shall see in a moment.

Step 2: A recursive solution

Theorem 15.1 implies that we should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whichever of these two LCSs is longer is an LCS of X and Y. Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must appear within an LCS of X and Y.

We can readily see the overlapping-subproblems property in the LCS problem. To find an LCS of X and Y, we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}. Many other subproblems share subsubproblems.

As in the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a recurrence for the value of an optimal solution. Let us define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases} \qquad (15.9)$$

Observe that in this recursive formulation, a condition in the problem restricts which subproblems we may consider. When $x_i = y_j$, we can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}. Otherwise, we instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j. In the previous dynamic-programming algorithms we have examined—for rod cutting and matrix-chain multiplication—we ruled out no subproblems due to conditions in the problem. Finding an LCS is not the only dynamic-programming algorithm that rules out subproblems based on conditions in the problem. For example, the edit-distance problem (see Problem 15-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (15.9), we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since the LCS problem has only $\Theta(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

Procedure LCS-LENGTH takes two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as inputs. It stores the $c[i, j]$ values in a table $c[0 \mathinner{..} m, 0 \mathinner{..} n]$, and it computes the entries in ***row-major*** order. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.) The procedure also maintains the table $b[1 \mathinner{..} m, 1 \mathinner{..} n]$ to help us construct an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns the b and c tables; $c[m, n]$ contains the length of an LCS of X and Y.

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

Figure 15.8 shows the tables produced by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Step 4: Constructing an LCS

The b table returned by LCS-LENGTH enables us to quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. We simply begin at $b[m, n]$ and trace through the table by following the arrows. Whenever we encounter a "↖" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

found. With this method, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS$(b, X, X.\mathit{length}, Y.\mathit{length})$.

```
PRINT-LCS$(b, X, i, j)$
1  **if** $i == 0$ or $j == 0$
2      **return**
3  **if** $b[i, j] ==$ "↖"
4      PRINT-LCS$(b, X, i-1, j-1)$
5      print $x_i$
6  **elseif** $b[i, j] ==$ "↑"
7      PRINT-LCS$(b, X, i-1, j)$
8  **else** PRINT-LCS$(b, X, i, j-1)$
```

For the b table in Figure 15.8, this procedure prints $BCBA$. The procedure takes time $O(m + n)$, since it decrements at least one of i and j in each recursive call.

Improving the code

Once you have developed an algorithm, you will often find that you can improve on the time or space it uses. Some changes can simplify the code and improve constant factors but otherwise yield no asymptotic improvement in performance. Others can yield substantial asymptotic savings in time and space.

In the LCS algorithm, for example, we can eliminate the b table altogether. Each $c[i, j]$ entry depends on only three other c table entries: $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given the value of $c[i, j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i, j]$, without inspecting table b. Thus, we can reconstruct an LCS in $O(m+n)$ time using a procedure similar to PRINT-LCS. (Exercise 15.4-2 asks you to give the pseudocode.) Although we save $\Theta(mn)$ space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we need $\Theta(mn)$ space for the c table anyway.

We can, however, reduce the asymptotic space requirements for LCS-LENGTH, since it needs only two rows of table c at a time: the row being computed and the previous row. (In fact, as Exercise 15.4-4 asks you to show, we can use only slightly more than the space for one row of c to compute the length of an LCS.) This improvement works if we need only the length of an LCS; if we need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace our steps in $O(m + n)$ time.

Exercises

15.4-1

Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

15.4-3

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

15.4-4

Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

15.4-5

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

15.4-6 $\star$

Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

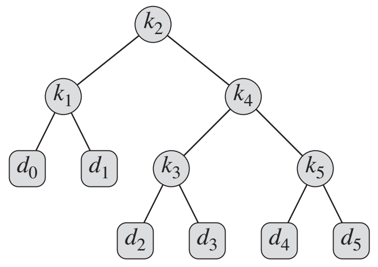
15.5 Optimal binary search trees

Suppose that we are designing a program to translate text from English to French. For each occurrence of each English word in the text, we need to look up its French equivalent. We could perform these lookup operations by building a binary search tree with n English words as keys and their French equivalents as satellite data. Because we will search the tree for each individual word in the text, we want the total time spent searching to be as low as possible. We could ensure an $O(\lg n)$ search time per occurrence by using a red-black tree or any other balanced binary search tree. Words appear with different frequencies, however, and a frequently used word such as *the* may appear far from the root while a rarely used word such as *machicolation* appears near the root. Such an organization would slow down the translation, since the number of nodes visited when searching for a key in a binary search tree equals one plus the depth of the node containing the key. We want words that occur frequently in the text to be placed nearer the root.[6] Moreover, some words in the text might have no French translation,[7] and such words would not appear in the binary search tree at all. How do we organize a binary search tree so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs?

What we need is known as an ***optimal binary search tree***. Formally, we are given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order (so that $k_1 < k_2 < \cdots < k_n$), and we wish to build a binary search tree from these keys. For each key k_i, we have a probability p_i that a search will be for k_i. Some searches may be for values not in K, and so we also have $n + 1$ "dummy keys"

[6] If the subject of the text is castle architecture, we might want *machicolation* to appear near the root.

[7] Yes, *machicolation* has a French counterpart: *mâchicoulis*.



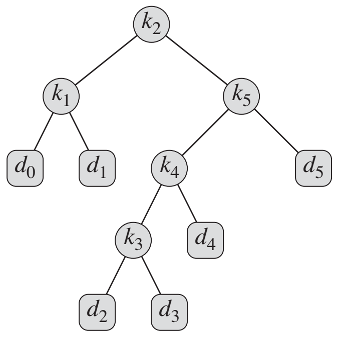


Figure 15.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.

$d_0, d_1, d_2, \ldots, d_n$ representing values not in K. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have a probability q_i that a search will correspond to d_i. Figure 15.9 shows two binary search trees for a set of $n = 5$ keys. Each key k_i is an internal node, and each dummy key d_i is a leaf. Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \tag{15.10}$$

Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given binary search tree T. Let us assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is

$$\begin{aligned} \text{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i , \qquad (15.11) \end{aligned}$$

where depth_T denotes a node's depth in the tree T. The last equality follows from equation (15.10). In Figure 15.9(a), we can calculate the expected search cost node by node:

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is smallest. We call such a tree an ***optimal binary search tree***. Figure 15.9(b) shows an optimal binary search tree for the probabilities given in the figure caption; its expected cost is 2.75. This example shows that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Nor can we necessarily construct an optimal binary search tree by always putting the key with the greatest probability at the root. Here, key k_5 has the greatest search probability of any key, yet the root of the optimal binary search tree shown is k_2. (The lowest expected cost of any binary search tree with k_5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm. We can label the nodes of any n-node binary tree with the keys $k_1, k_2, \ldots, k_n$ to construct a binary search tree, and then add in the dummy keys as leaves. In Problem 12-4, we saw that the number of binary trees with n nodes is $\Omega(4^n/n^{3/2})$, and so we would have to examine an exponential number of binary search trees in an exhaustive search. Not surprisingly, we shall solve this problem with dynamic programming.

Step 1: The structure of an optimal binary search tree

To characterize the optimal substructure of optimal binary search trees, we start with an observation about subtrees. Consider any subtree of a binary search tree. It must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \leq i \leq j \leq n$. In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.

Now we can state the optimal substructure: if an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. The usual cut-and-paste argument applies. If there were a subtree T'' whose expected cost is lower than that of T', then we could cut T' out of T and paste in T'', resulting in a binary search tree of lower expected cost than T, thus contradicting the optimality of T.

We need to use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. Given keys $k_i, \ldots, k_j$, one of these keys, say k_r ($i \le r \le j$), is the root of an optimal subtree containing these keys. The left subtree of the root k_r contains the keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$), and the right subtree contains the keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$). As long as we examine all candidate roots k_r, where $i \le r \le j$, and we determine all optimal binary search trees containing $k_i, \ldots, k_{r-1}$ and those containing $k_{r+1}, \ldots, k_j$, we are guaranteed that we will find an optimal binary search tree.

There is one detail worth noting about "empty" subtrees. Suppose that in a subtree with keys $k_i, \ldots, k_j$, we select k_i as the root. By the above argument, k_i's left subtree contains the keys $k_i, \ldots, k_{i-1}$. We interpret this sequence as containing no keys. Bear in mind, however, that subtrees also contain dummy keys. We adopt the convention that a subtree containing keys $k_i, \ldots, k_{i-1}$ has no actual keys but does contain the single dummy key d_{i-1}. Symmetrically, if we select k_j as the root, then k_j's right subtree contains the keys $k_{j+1}, \ldots, k_j$; this right subtree contains no actual keys, but it does contain the dummy key d_j.

Step 2: A recursive solution

We are ready to define the value of an optimal solution recursively. We pick our subproblem domain as finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \ge 1$, $j \le n$, and $j \ge i - 1$. (When $j = i - 1$, there are no actual keys; we have just the dummy key d_{i-1}.) Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Ultimately, we wish to compute $e[1, n]$.

The easy case occurs when $j = i - 1$. Then we have just the dummy key d_{i-1}. The expected search cost is $e[i, i-1] = q_{i-1}$.

When $j \ge i$, we need to select a root k_r from among $k_i, \ldots, k_j$ and then make an optimal binary search tree with keys $k_i, \ldots, k_{r-1}$ as its left subtree and an optimal binary search tree with keys $k_{r+1}, \ldots, k_j$ as its right subtree. What happens to the expected search cost of a subtree when it becomes a subtree of a node? The depth of each node in the subtree increases by 1. By equation (15.11), the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For a subtree with keys $k_i, \ldots, k_j$, let us denote this sum of probabilities as

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \; . \tag{15.12}$$

Thus, if k_r is the root of an optimal subtree containing keys $k_i, \ldots, k_j$, we have

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)) \; .$$

Noting that

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j) \; ,$$

we rewrite $e[i,j]$ as

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) \; . \tag{15.13}$$

The recursive equation (15.13) assumes that we know which node k_r to use as the root. We choose the root that gives the lowest expected search cost, giving us our final recursive formulation:

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \; , \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \; . \end{cases} \tag{15.14}$$

The $e[i,j]$ values give the expected search costs in optimal binary search trees. To help us keep track of the structure of optimal binary search trees, we define *root*$[i,j]$, for $1 \le i \le j \le n$, to be the index r for which k_r is the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. Although we will see how to compute the values of *root*$[i,j]$, we leave the construction of an optimal binary search tree from these values as Exercise 15.5-1.

Step 3: Computing the expected search cost of an optimal binary search tree

At this point, you may have noticed some similarities between our characterizations of optimal binary search trees and matrix-chain multiplication. For both problem domains, our subproblems consist of contiguous index subranges. A direct, recursive implementation of equation (15.14) would be as inefficient as a direct, recursive matrix-chain multiplication algorithm. Instead, we store the $e[i,j]$ values in a table $e[1\,..\,n+1, 0\,..\,n]$. The first index needs to run to $n+1$ rather than n because in order to have a subtree containing only the dummy key d_n, we need to compute and store $e[n+1,n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0, we need to compute and store $e[1,0]$. We use only the entries $e[i,j]$ for which $j \ge i-1$. We also use a table *root*$[i,j]$, for recording the root of the subtree containing keys $k_i, \ldots, k_j$. This table uses only the entries for which $1 \le i \le j \le n$.

We will need one other table for efficiency. Rather than compute the value of $w(i,j)$ from scratch every time we are computing $e[i,j]$—which would take

$\Theta(j-i)$ additions—we store these values in a table $w[1\,..\,n+1, 0\,..\,n]$. For the base case, we compute $w[i, i-1] = q_{i-1}$ for $1 \le i \le n+1$. For $j \ge i$, we compute

$$w[i, j] = w[i, j-1] + p_j + q_j \; . \tag{15.15}$$

Thus, we can compute the $\Theta(n^2)$ values of $w[i, j]$ in $\Theta(1)$ time each.

The pseudocode that follows takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and it returns the tables e and *root*.

```
OPTIMAL-BST(p, q, n)
 1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
            and root[1..n, 1..n] be new tables
 2  for i = 1 to n + 1
 3      e[i, i-1] = q_{i-1}
 4      w[i, i-1] = q_{i-1}
 5  for l = 1 to n
 6      for i = 1 to n - l + 1
 7          j = i + l - 1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j-1] + p_j + q_j
10          for r = i to j
11              t = e[i, r-1] + e[r+1, j] + w[i, j]
12              if t < e[i, j]
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

From the description above and the similarity to the MATRIX-CHAIN-ORDER procedure in Section 15.2, you should find the operation of this procedure to be fairly straightforward. The **for** loop of lines 2–4 initializes the values of $e[i, i-1]$ and $w[i, i-1]$. The **for** loop of lines 5–14 then uses the recurrences (15.14) and (15.15) to compute $e[i, j]$ and $w[i, j]$ for all $1 \le i \le j \le n$. In the first iteration, when $l = 1$, the loop computes $e[i, i]$ and $w[i, i]$ for $i = 1, 2, \ldots, n$. The second iteration, with $l = 2$, computes $e[i, i+1]$ and $w[i, i+1]$ for $i = 1, 2, \ldots, n-1$, and so forth. The innermost **for** loop, in lines 10–14, tries each candidate index r to determine which key k_r to use as the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. This **for** loop saves the current value of the index r in *root*$[i, j]$ whenever it finds a better key to use as the root.

Figure 15.10 shows the tables $e[i, j]$, $w[i, j]$, and *root*$[i, j]$ computed by the procedure OPTIMAL-BST on the key distribution shown in Figure 15.9. As in the matrix-chain multiplication example of Figure 15.5, the tables are rotated to make

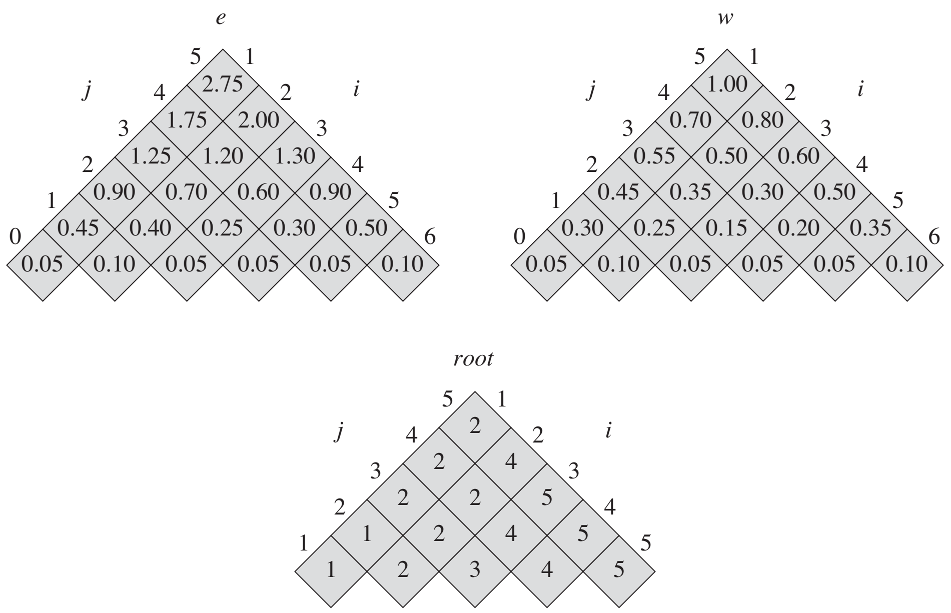

e
j
i
2.75
1.75 2.00
1.25 1.20 1.30
0.90 0.70 0.60 0.90
0.45 0.40 0.25 0.30 0.50
0.05 0.10 0.05 0.05 0.05 0.10
w
j
i
1.00
0.70 0.80
0.55 0.50 0.60
0.45 0.35 0.30 0.50
0.30 0.25 0.15 0.20 0.35
0.05 0.10 0.05 0.05 0.05 0.10
root
j
i
2
2 4
2 2 5
1 2 4 5
1 2 3 4 5

Figure 15.10 The tables $e[i, j]$, $w[i, j]$, and *root*$[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 15.9. The tables are rotated so that the diagonals run horizontally.

the diagonals run horizontally. OPTIMAL-BST computes the rows from bottom to top and from left to right within each row.

The OPTIMAL-BST procedure takes $\Theta(n^3)$ time, just like MATRIX-CHAIN-ORDER. We can easily see that its running time is $O(n^3)$, since its **for** loops are nested three deep and each loop index takes on at most n values. The loop indices in OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-ORDER, but they are within at most 1 in all directions. Thus, like MATRIX-CHAIN-ORDER, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time.

Exercises

15.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST(*root*) which, given the table *root*, outputs the structure of an optimal binary search tree. For the example in Figure 15.10, your procedure should print out the structure

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 15.9(b).

15.5-2
Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

15.5-3
Suppose that instead of maintaining the table $w[i, j]$, we computed the value of $w(i, j)$ directly from equation (15.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

15.5-4 ★
Knuth [212] has shown that there are always roots of optimal subtrees such that $root[i, j-1] \le root[i, j] \le root[i+1, j]$ for all $1 \le i < j \le n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.

Problems

15-1 Longest simple path in a directed acyclic graph
Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?

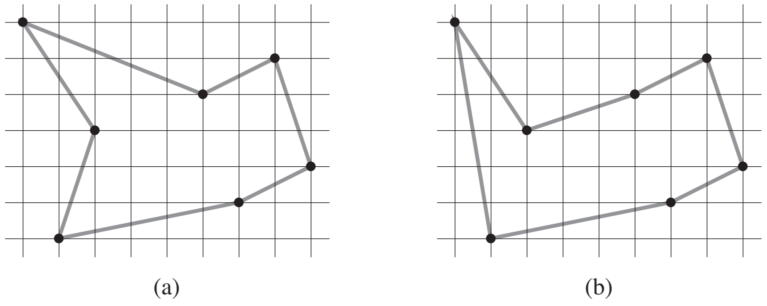


Figure 15.11 Seven points in the plane, shown on a unit grid. **(a)** The shortest closed tour, with length approximately 24.89. This tour is not bitonic. **(b)** The shortest bitonic tour for the same set of points. Its length is approximately 25.58.

15-2 Longest palindrome subsequence

A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

15-3 Bitonic euclidean traveling-salesman problem

In the ***euclidean traveling-salesman problem***, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).

J. L. Bentley has suggested that we simplify the problem by restricting our attention to ***bitonic tours***, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. (*Hint:* Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

15-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n

words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \le j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

15-5 Edit distance

In order to transform one source string of text $x[1\,..\,m]$ to a target string $y[1\,..\,n]$, we can perform various transformation operations. Our goal is, given x and y, to produce a series of transformations that change x to y. We use an array z—assumed to be large enough to hold all the characters it will need—to hold the intermediate results. Initially, z is empty, and at termination, we should have $z[j] = y[j]$ for $j = 1, 2, \ldots, n$. We maintain current indices i into x and j into z, and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. We are required to examine every character in x during the transformation, which means that at the end of the sequence of transformation operations, we must have $i = m + 1$.

We may choose from among six transformation operations:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$.

Replace a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$.

Insert the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x.

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order; we do so by setting $z[j] = x[i+1]$ and $z[j+1] = x[i]$ and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i+1]$.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation.

As an example, one way to transform the source string `algorithm` to the target string `altruistic` is to use the following sequence of operations, where the underlined characters are $x[i]$ and $z[j]$ after the operation:

Operation	x	z
initial strings	$\texttt{\underline{a}lgorithm}$	$\texttt{\_}$
copy	$\texttt{a\underline{l}gorithm}$	$\texttt{a\_}$
copy	$\texttt{al\underline{g}orithm}$	$\texttt{al\_}$
replace by `t`	$\texttt{alg\underline{o}rithm}$	$\texttt{alt\_}$
delete	$\texttt{algo\underline{r}ithm}$	$\texttt{alt\_}$
copy	$\texttt{algor\underline{i}thm}$	$\texttt{altr\_}$
insert `u`	$\texttt{algor\underline{i}thm}$	$\texttt{altru\_}$
insert `i`	$\texttt{algor\underline{i}thm}$	$\texttt{altrui\_}$
insert `s`	$\texttt{algor\underline{i}thm}$	$\texttt{altruis\_}$
twiddle	$\texttt{algorit\underline{h}m}$	$\texttt{altruisti\_}$
insert `c`	$\texttt{algorit\underline{h}m}$	$\texttt{altruistic\_}$
kill	$\texttt{algorithm\_}$	$\texttt{altruistic\_}$

Note that there are several other sequences of transformation operations that transform `algorithm` to `altruistic`.

Each of the transformation operations has an associated cost. The cost of an operation depends on the specific application, but we assume that each operation's cost is a constant that is known to us. We also assume that the individual costs of the copy and replace operations are less than the combined costs of the delete and insert operations; otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming `algorithm` to `altruistic` is

$$\begin{aligned}&(3 \cdot \text{cost(copy)}) + \text{cost(replace)} + \text{cost(delete)} + (4 \cdot \text{cost(insert)}) \\ &\quad + \text{cost(twiddle)} + \text{cost(kill)}\ .\end{aligned}$$

a. Given two sequences $x[1\,..\,m]$ and $y[1\,..\,n]$ and set of transformation-operation costs, the ***edit distance*** from x to y is the cost of the least expensive operation sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from $x[1\,..\,m]$ to $y[1\,..\,n]$ and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [310, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a "score" to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x =$ GATCGGCAT and $y =$ CAATGTGAATC, one alignment is

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

A + under a position indicates a score of $+1$ for that position, a - indicates a score of -1, and a * indicates a score of -2, so that this alignment has a total score of $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

15-6 Planning a company party

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

15-7 Viterbi algorithm

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. We define the label of a directed path to be the concatenation of the labels of the edges on that path.

a. Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ of sounds from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. (*Hint:* You may find concepts from Chapter 22 useful.)

Now, suppose that every edge $(u, v) \in E$ has an associated nonnegative probability $p(u, v)$ of traversing the edge (u, v) from vertex u and thus producing the corresponding sound. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. We can view the probability of a path beginning at v_0 as the probability that a "random walk" beginning at v_0 will follow the specified path, where we randomly choose which edge to take leaving a vertex u according to the probabilities of the available edges leaving u.

b. Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at v_0 and having label s. Analyze the running time of your algorithm.

15-8 Image compression by seam carving

We are given a color picture consisting of an $m \times n$ array $A[1\,..\,m, 1\,..\,n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. Suppose that we wish to compress this picture slightly. Specifically, we wish to remove one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns; the pixels removed form a "seam" from the top row to the bottom row where successive pixels in the seam are adjacent vertically or diagonally.

a. Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

b. Suppose now that along with each pixel $A[i, j]$, we have calculated a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

15-9 Breaking a string

A certain string-processing language allows a programmer to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose a programmer wants to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that the programmer wants to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If she programs the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If she programs the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, she could break first at 8 (costing 20), then break the left piece at 2 (costing 8), and finally the right piece at 10 (costing 12), for a total cost of 40.

Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given a string S with n characters and an array $L[1\,..\,m]$ containing the break points, compute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

15-10 Planning an investment strategy

Your knowledge of algorithms helps you obtain an exciting job with the Acme Computer Company, along with a \$10,000 signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use the Amalgamated Investment Company to manage your investments. Amalgamated Investments requires you to observe the following rules. It offers n different investments, numbered 1 through n. In each year j, investment i provides a return rate of r_{ij}. In other words, if you invest d dollars in investment i in year j, then at the end of year j, you have dr_{ij} dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investement. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$.

a. The problem, as stated, allows you to invest your money in multiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)

b. Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.

c. Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm?

d. Suppose that Amalgamated Investments imposed the additional restriction that, at any point, you can have no more than \$15,000 in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

15-11 Inventory planning

The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i, the company knows the demand d_i, that is, the number of machines that it will sell. Let $D = \sum_{i=1}^{n} d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if, at the end of a month, the company is holding any unsold machines, it must pay inventory costs. The cost for holding j machines is given as a function $h(j)$ for $j = 1, 2, \ldots, D$, where $h(j) \geq 0$ for $1 \leq j \leq D$ and $h(j) \leq h(j+1)$ for $1 \leq j \leq D-1$.

Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polyomial in n and D.

15-12 Signing free-agent baseball players

Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of \$$X$ to spend on free agents. You are allowed to spend less than \$$X$ altogether, but the owner will fire you if you spend any more than \$$X$.

You are considering N different positions, and for each position, P free-agent players who play that position are available.[8] Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)

To determine how valuable a player is going to be, you decide to use a sabermetric statistic[9] known as "VORP," or "value over replacement player." A player with a higher VORP is more valuable than a player with a lower VORP. A player with a higher VORP is not necessarily more expensive to sign than a player with a lower VORP, because factors other than a player's value determine how much it costs to sign him.

For each available free-agent player, you have three pieces of information:

- the player's position,
- the amount of money it will cost to sign the player, and
- the player's VORP.

Devise an algorithm that maximizes the total VORP of the players you sign while spending no more than $\$X$ altogether. You may assume that each player signs for a multiple of \$100,000. Your algorithm should output the total VORP of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

Chapter notes

R. Bellman began the systematic study of dynamic programming in 1955. The word "programming," both here and in linear programming, refers to using a tabular solution method. Although optimization techniques incorporating elements of dynamic programming were known earlier, Bellman provided the area with a solid mathematical basis [37].

[8] Although there are nine positions on a baseball team, N is not necesarily equal to 9 because some general managers have particular ways of thinking about positions. For example, a general manager might consider right-handed pitchers and left-handed pitchers to be separate "positions," as well as starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief pitchers (relief pitchers who normally pitch at most only one inning).

[9] ***Sabermetrics*** is the application of statistical analysis to baseball records. It provides several ways to compare the relative values of individual players.

Galil and Park [125] classify dynamic-programming algorithms according to the size of the table and the number of other table entries each entry depends on. They call a dynamic-programming algorithm tD/eD if its table size is $O(n^t)$ and each entry depends on $O(n^e)$ other entries. For example, the matrix-chain multiplication algorithm in Section 15.2 would be $2D/1D$, and the longest-common-subsequence algorithm in Section 15.4 would be $2D/0D$.

Hu and Shing [182, 183] give an $O(n \lg n)$-time algorithm for the matrix-chain multiplication problem.

The $O(mn)$-time algorithm for the longest-common-subsequence problem appears to be a folk algorithm. Knuth [70] posed the question of whether subquadratic algorithms for the LCS problem exist. Masek and Paterson [244] answered this question in the affirmative by giving an algorithm that runs in $O(mn/\lg n)$ time, where $n \le m$ and the sequences are drawn from a set of bounded size. For the special case in which no element appears more than once in an input sequence, Szymanski [326] shows how to solve the problem in $O((n+m)\lg(n+m))$ time. Many of these results extend to the problem of computing string edit distances (Problem 15-5).

An early paper on variable-length binary encodings by Gilbert and Moore [133] had applications to constructing optimal binary search trees for the case in which all probabilities p_i are 0; this paper contains an $O(n^3)$-time algorithm. Aho, Hopcroft, and Ullman [5] present the algorithm from Section 15.5. Exercise 15.5-4 is due to Knuth [212]. Hu and Tucker [184] devised an algorithm for the case in which all probabilities p_i are 0 that uses $O(n^2)$ time and $O(n)$ space; subsequently, Knuth [211] reduced the time to $O(n \lg n)$.

Problem 15-8 is due to Avidan and Shamir [27], who have posted on the Web a wonderful video illustrating this image-compression technique.

Greedy Algorithms

16 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 15, particularly Section 15.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We shall first examine, in Section 16.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We shall arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that we can always make greedy choices to arrive at an optimal solution. Section 16.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 16.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. In Section 16.4, we investigate some of the theory underlying combinatorial structures called "matroids," for which a greedy algorithm always produces an optimal solution. Finally, Section 16.5 applies matroids to solve a problem of scheduling unit-time tasks with deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that we can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 23), Dijkstra's algorithm for shortest paths from a single source (Chapter 24), and Chvátal's greedy set-covering heuristic (Chapter 35). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read this chapter and Chapter 23 independently of each other, you might find it useful to read them together.

16.1 An activity-selection problem

Our first example is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \le s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$. In the ***activity-selection problem***, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n . \tag{16.1}$$

(We shall see later the advantage that this assumption provides.) For example, consider the following set S of activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We shall solve this problem in several steps. We start by thinking about a dynamic-programming solution, in which we consider several choices when determining which subproblems to use in an optimal solution. We shall then observe that we need to consider only one choice—the greedy choice—and that when we make the greedy choice, only one subproblem remains. Based on these observations, we shall develop a recursive greedy algorithm to solve the activity-scheduling problem. We shall complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. Although the steps we shall go through in this section are slightly more involved than is typical when developing a greedy algorithm, they illustrate the relationship between greedy algorithms and dynamic programming.

The optimal substructure of the activity-selection problem

We can easily verify that the activity-selection problem exhibits optimal substructure. Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij}, which includes some activity a_k. By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts). Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then we could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. We would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik}.

This way of characterizing optimal substructure suggests that we might solve the activity-selection problem by dynamic programming. If we denote the size of an optimal solution for the set S_{ij} by $c[i,j]$, then we would have the recurrence

$$c[i,j] = c[i,k] + c[k,j] + 1 .$$

Of course, if we did not know that an optimal solution for the set S_{ij} includes activity a_k, we would have to examine all activities in S_{ij} to find which one to choose, so that

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset . \end{cases} \tag{16.2}$$

We could then develop a recursive algorithm and memoize it, or we could work bottom-up and fill in table entries as we go along. But we would be overlooking another important characteristic of the activity-selection problem that we can use to great advantage.

Making the greedy choice

What if we could choose an activity to add to our optimal solution without having to first solve all the subproblems? That could save us from having to consider all the choices inherent in recurrence (16.2). In fact, for the activity-selection problem, we need consider only one choice: the greedy choice.

What do we mean by the greedy choice for the activity-selection problem? Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible. Now, of the activities we end up choosing, one of them must be the first one to finish. Our intuition tells us, therefore, to choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible. (If more than one activity in S has the earliest finish time, then we can choose any such activity.) In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1. Choosing the first activity to finish is not the only way to think of making a greedy choice for this problem; Exercise 16.1-3 asks you to explore other possibilities.

If we make the greedy choice, we have only one remaining subproblem to solve: finding activities that start after a_1 finishes. Why don't we have to consider activities that finish before a_1 starts? We have that $s_1 < f_1$, and f_1 is the earliest finish time of any activity, and therefore no activity can have a finish time less than or equal to s_1. Thus, all activities that are compatible with activity a_1 must start after a_1 finishes.

Furthermore, we have already established that the activity-selection problem exhibits optimal substructure. Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If we make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.[1] Optimal substructure tells us that if a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1.

One big question remains: is our intuition correct? Is the greedy choice—in which we choose the first activity to finish—always part of some optimal solution? The following theorem shows that it is.

[1]We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. It will always be clear from the context whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem 16.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

Thus, we see that although we might be able to solve the activity-selection problem with dynamic programming, we don't need to. (Besides, we have not yet examined whether the activity-selection problem even has overlapping subproblems.) Instead, we can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain. Moreover, because we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase. We can consider each activity just once overall, in monotonically increasing order of finish times.

An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm. Instead, it can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen. Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Now that we have seen how to bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, we can write a straightforward, recursive procedure to solve the activity-selection problem. The procedure RECURSIVE-ACTIVITY-SELECTOR takes the start and finish times of the activities, represented as arrays s and f,[2] the index k that defines the subproblem S_k it is to solve, and

[2]Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather than subscripts.

the size n of the original problem. It returns a maximum-size set of mutually compatible activities in S_k. We assume that the n input activities are already ordered by monotonically increasing finish time, according to equation (16.1). If not, we can sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily. In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S. The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  $m = k + 1$
2  while $m \le n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
3      $m = m + 1$
4  if $m \le n$
5      return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
6  else return $\emptyset$
```

Figure 16.1 shows the operation of the algorithm. In a given recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n), the **while** loop of lines 2–3 looks for the first activity in S_k to finish. The loop examines $a_{k+1}, a_{k+2}, \ldots, a_n$, until it finds the first activity a_m that is compatible with a_k; such an activity has $s_m \ge f_k$. If the loop terminates because it finds such an activity, line 5 returns the union of $\{a_m\}$ and the maximum-size subset of S_m returned by the recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n). Alternatively, the loop may terminate because $m > n$, in which case we have examined all activities in S_k without finding one that is compatible with a_k. In this case, $S_k = \emptyset$, and so the procedure returns $\emptyset$ in line 6.

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ is $\Theta(n)$, which we can see as follows. Over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular, activity a_i is examined in the last call made in which $k < i$.

An iterative greedy algorithm

We easily can convert our recursive procedure to an iterative one. The procedure RECURSIVE-ACTIVITY-SELECTOR is almost "tail recursive" (see Problem 7-4): it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form; in fact, some compilers for certain programming languages perform this task automatically. As written, RECURSIVE-ACTIVITY-SELECTOR works for subproblems S_k, i.e., subproblems that consist of the last activities to finish.

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16

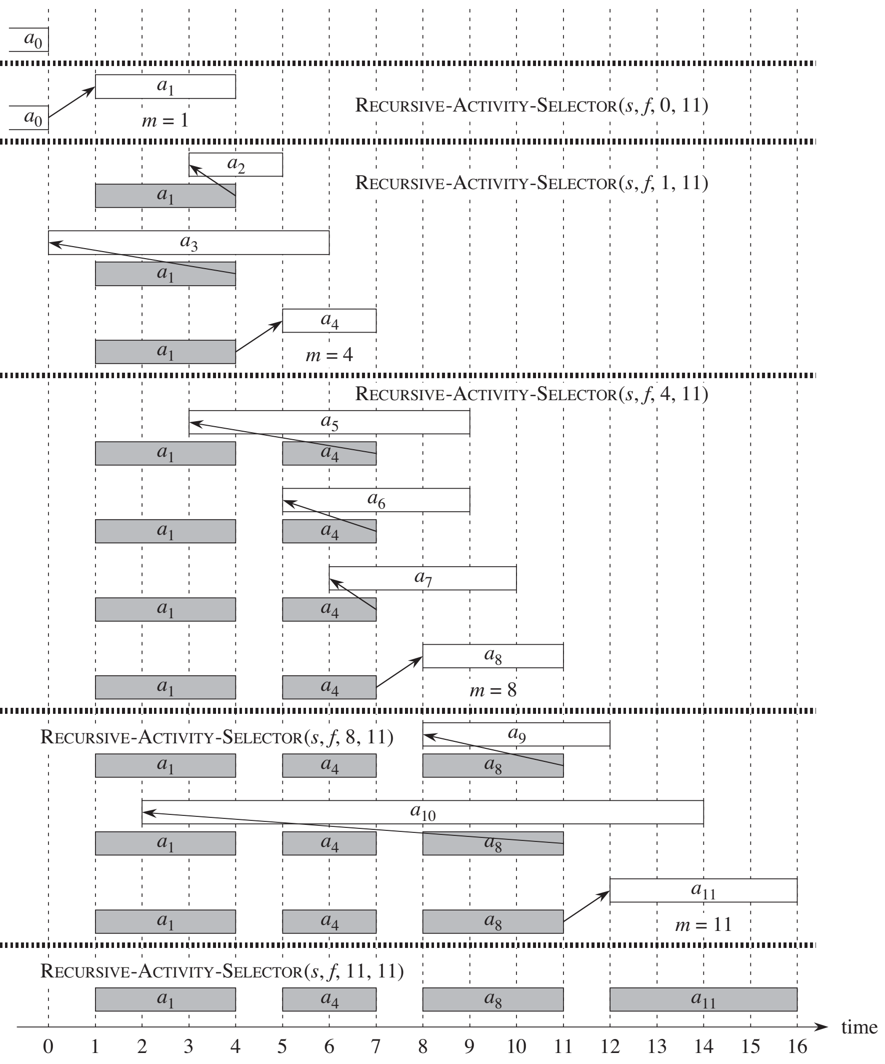

a_0
a_1
$m = 1$
RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, 11)$
a_2
RECURSIVE-ACTIVITY-SELECTOR$(s, f, 1, 11)$
a_3
a_4
$m = 4$
RECURSIVE-ACTIVITY-SELECTOR$(s, f, 4, 11)$
a_5
a_6
a_7
a_8
$m = 8$
RECURSIVE-ACTIVITY-SELECTOR$(s, f, 8, 11)$
a_9
a_{10}
a_{11}
$m = 11$
RECURSIVE-ACTIVITY-SELECTOR$(s, f, 11, 11)$
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figure 16.1 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities given earlier. Activities considered in each recursive call appear between horizontal lines. The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, 11)$, selects activity a_1. In each recursive call, the activities that have already been selected are shaded, and the activity shown in white is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, RECURSIVE-ACTIVITY-SELECTOR$(s, f, 11, 11)$, returns $\emptyset$. The resulting set of selected activities is $\{a_1, a_4, a_8, a_{11}\}$.

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR. It also assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set A and returns this set when it is done.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

The procedure works as follows. The variable k indexes the most recent addition to A, corresponding to the activity a_k in the recursive version. Since we consider the activities in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A. That is,

$$f_k = \max\{f_i : a_i \in A\} . \tag{16.3}$$

Lines 2–3 select activity a_1, initialize A to contain just this activity, and initialize k to index this activity. The **for** loop of lines 4–7 finds the earliest activity in S_k to finish. The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities; such an activity is the earliest in S_k to finish. To see whether activity a_m is compatible with every activity currently in A, it suffices by equation (16.3) to check (in line 5) that its start time s_m is not earlier than the finish time f_k of the activity most recently added to A. If activity a_m is compatible, then lines 6–7 add activity a_m to A and set k to m. The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Exercises

16.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.

Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

16.1-2

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

16.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

16.1-4

Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the ***interval-graph coloring problem***. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

16.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

16.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in Section 16.1 to develop a greedy algorithm was a bit more involved than is typical. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (16.2), but we bypassed developing a recursive algorithm based on this recurrence.)
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

In going through these steps, we saw in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, in the activity-selection problem, we first defined the subproblems S_{ij}, where both i and j varied. We then found that if we always made the greedy choice, we could restrict the subproblems to be of the form S_k.

Alternatively, we could have fashioned our optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, we could have started by dropping the second subscript and defining subproblems of the form S_k. Then, we could have proven that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k. More generally, we design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

We shall use this more direct process in later sections of this chapter. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

How can we tell whether a greedy algorithm will solve a particular optimization problem? No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients. If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the ***greedy-choice property***: we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. (Alternatively, we can solve them top down, but memoizing. Of course, even though the code works top down, we still must solve the subproblems before making a choice.) In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, we must prove that a greedy choice at each step yields a globally optimal solution. Typically, as in the case of Theorem 16.1, the proof examines a globally optimal solution to some subproblem. It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

We can usually make the greedy choice more efficiently than when we have to consider a wider set of choices. For example, in the activity-selection problem, assuming that we had already sorted the activities in monotonically increasing order of finish times, we needed to examine each activity just once. By preprocessing the input or by using an appropriate data structure (often a priority queue), we often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. As an example of optimal substructure, recall how we demonstrated in Section 16.1 that if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj}. Given this optimal substructure, we argued that if we knew which activity to use as a_k, we could construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the subproblems S_{ik} and S_{kj}. Based on this observation of optimal substructure, we were able to devise the recurrence (16.2) that described the value of an optimal solution.

We usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, we have the luxury of assuming that we arrived at a subproblem by having made the greedy choice in the original problem. All we really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem*** is the following. A thief robbing a store finds n items. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j. For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy. To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W. Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. We leave the proof that the fractional knapsack problem has the greedy-choice property as Exercise 16.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 16.2(a). This example has 3 items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is 6 dollars per pound, which is greater than the value per pound of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 16.2(b), however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two possible solutions that take item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 16.2(c). Taking item 1 doesn't work in the 0-1 problem because the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the

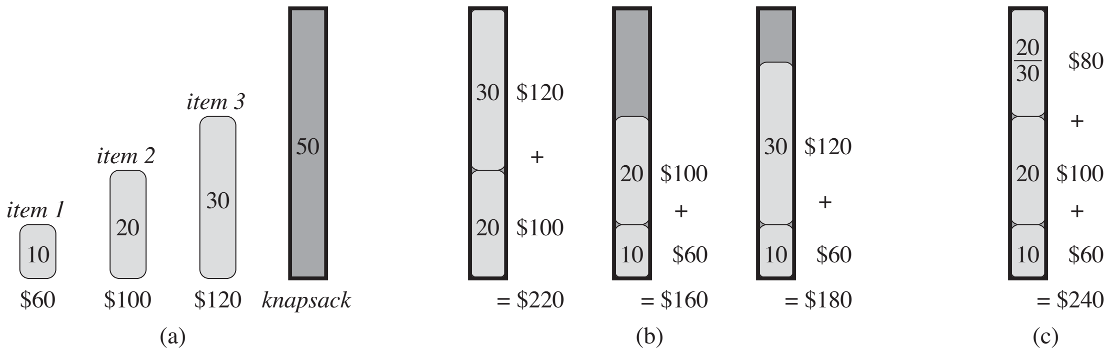


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

choice. The problem formulated in this way gives rise to many overlapping subproblems—a hallmark of dynamic programming, and indeed, as Exercise 16.2-2 asks you to show, we can use dynamic programming to solve the 0-1 problem.

Exercises

16.2-1
Prove that the fractional knapsack problem has the greedy-choice property.

16.2-2
Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(n\,W)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

16.2-3
Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

16.2-4
Professor Gekko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana.

The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations.

The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

16.2-5
Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

16.2-6 ★
Show how to solve the fractional knapsack problem in $O(n)$ time.

16.2-7
Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} {a_i}^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

16.3 Huffman codes

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by Figure 16.3. That is, only 6 different characters appear, and the character `a` occurs 45,000 times.

We have many options for how to represent such a file of information. Here, we consider the problem of designing a ***binary character code*** (or ***code*** for short)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters `a`–`f`, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

in which each character is represented by a unique binary string, which we call a ***codeword***. If we use a ***fixed-length code***, we need 3 bits to represent 6 characters: `a` = 000, `b` = 001, ..., `f` = 101. This method requires 300,000 bits to code the entire file. Can we do better?

A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Figure 16.3 shows such a code; here the 1-bit string 0 represents `a`, and the 4-bit string 1100 represents `f`. This code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix codes***.[3] Although we won't prove it here, a prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.

Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code of Figure 16.3, we code the 3-character file `abc` as $0 \cdot 101 \cdot 100 = 0101100$, where "·" denotes concatenation.

Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original char-

[3]Perhaps "prefix-free codes" would be a better name, but the term "prefix codes" is standard in the literature.

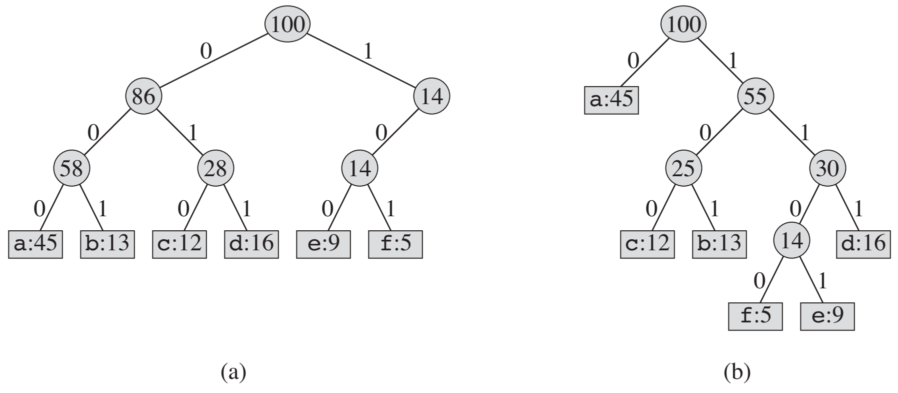


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code `a` = 000, ..., `f` = 101. **(b)** The tree corresponding to the optimal prefix code `a` = 0, `b` = 101, ..., `f` = 1100.

acter, and repeat the decoding process on the remainder of the encoded file. In our example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to `aabe`.

The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Figure 16.4 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 16.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 16.4(a), is not a full binary tree: it contains codewords beginning 10..., but none beginning 11.... Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3).

Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file. For each character c in the alphabet C, let the attribute $c.\mathit{freq}$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) , \tag{16.4}$$

which we define as the ***cost*** of the tree T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a ***Huffman code***. In line with our observations in Section 16.2, its proof of correctness relies on the greedy-choice property and optimal substructure. Rather than demonstrating that these properties hold and then developing pseudocode, we present the pseudocode first. Doing so will help clarify how the algorithm makes greedy choices.

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.\mathit{freq}$ giving its frequency. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN($C$)
1  $n = |C|$
2  $Q = C$
3  for $i = 1$ to $n - 1$
4      allocate a new node $z$
5      $z.left = x = $ EXTRACT-MIN($Q$)
6      $z.right = y = $ EXTRACT-MIN($Q$)
7      $z.freq = x.freq + y.freq$
8      INSERT($Q, z$)
9  return EXTRACT-MIN($Q$)      // return the root of the tree
```

For our example, Huffman's algorithm proceeds as shown in Figure 16.5. Since the alphabet contains 6 letters, the initial queue size is $n = 6$, and 5 merge steps build the tree. The final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

Line 2 initializes the min-priority queue Q with the characters in C. The **for** loop in lines 3–8 repeatedly extracts the two nodes x and y of lowest frequency

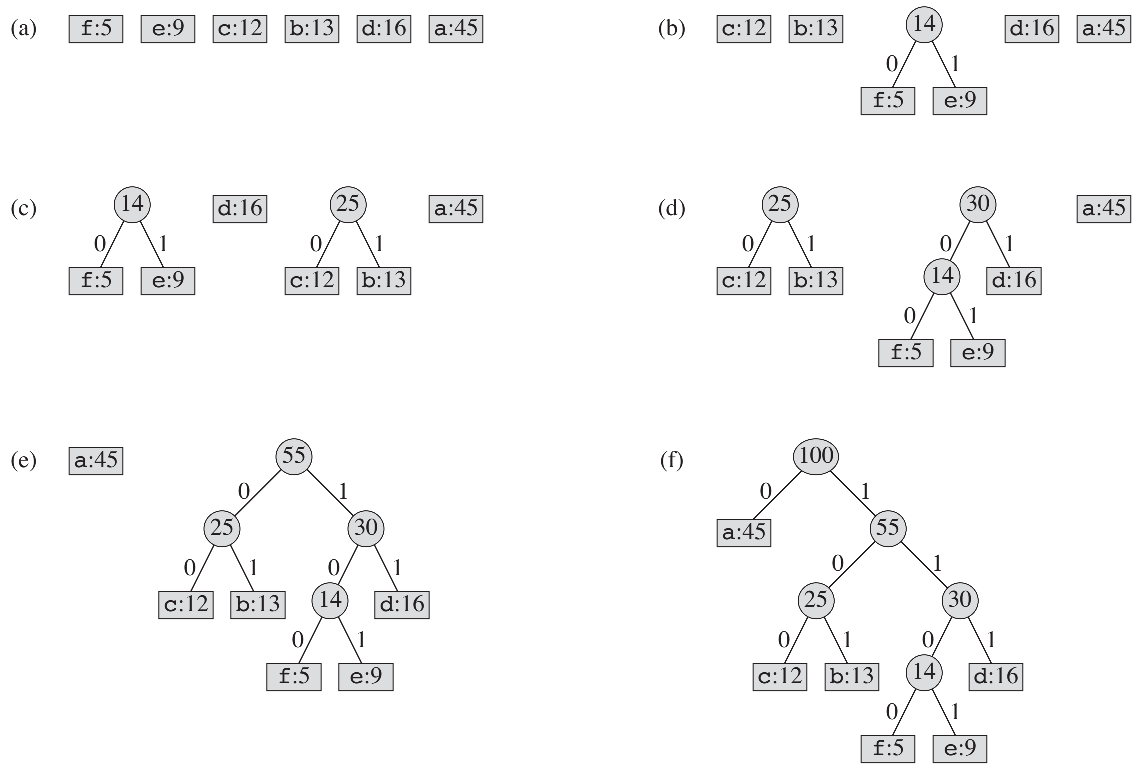


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

from the queue, replacing them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 7. The node z has x as its left child and y as its right child. (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, line 9 returns the one node left in the queue, which is the root of the code tree.

Although the algorithm would produce the same result if we were to excise the variables x and y—assigning directly to $z.left$ and $z.right$ in lines 5 and 6, and changing line 7 to $z.freq = z.left.freq + z.right.freq$—we shall use the node names x and y in the proof of correctness. Therefore, we find it convenient to leave them in.

To analyze the running time of Huffman's algorithm, we assume that Q is implemented as a binary min-heap (see Chapter 6). For a set C of n characters, we can initialize Q in line 2 in $O(n)$ time using the BUILD-MIN-HEAP procedure discussed in Section 6.3. The **for** loop in lines 3–8 executes exactly $n - 1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$. We can reduce the running time to $O(n \lg \lg n)$ by replacing the binary min-heap with a van Emde Boas tree (see Chapter 20).

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

Lemma 16.2
Let C be an alphabet in which each character $c \in C$ has frequency $c.\mathit{freq}$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.

Let a and b be two characters that are sibling leaves of maximum depth in T. Without loss of generality, we assume that $a.\mathit{freq} \le b.\mathit{freq}$ and $x.\mathit{freq} \le y.\mathit{freq}$. Since $x.\mathit{freq}$ and $y.\mathit{freq}$ are the two lowest leaf frequencies, in order, and $a.\mathit{freq}$ and $b.\mathit{freq}$ are two arbitrary frequencies, in order, we have $x.\mathit{freq} \le a.\mathit{freq}$ and $y.\mathit{freq} \le b.\mathit{freq}$.

In the remainder of the proof, it is possible that we could have $x.\mathit{freq} = a.\mathit{freq}$ or $y.\mathit{freq} = b.\mathit{freq}$. However, if we had $x.\mathit{freq} = b.\mathit{freq}$, then we would also have $a.\mathit{freq} = b.\mathit{freq} = x.\mathit{freq} = y.\mathit{freq}$ (see Exercise 16.3-1), and the lemma would be trivially true. Thus, we will assume that $x.\mathit{freq} \ne b.\mathit{freq}$, which means that $x \ne b$.

As Figure 16.6 shows, we exchange the positions in T of a and x to produce a tree T', and then we exchange the positions in T' of b and y to produce a tree T''

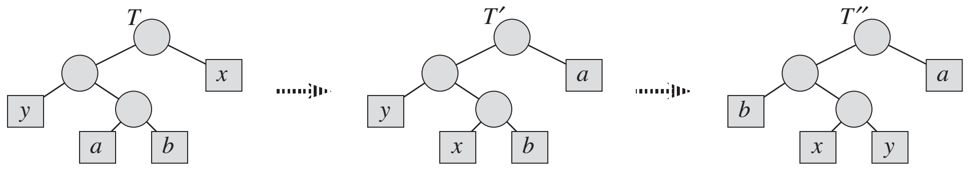


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (16.4), the difference in cost between T and T' is

$$\begin{aligned}
B(T) - B(T') & \\
&= \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) - \sum_{c \in C} c.\mathit{freq} \cdot d_{T'}(c) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_{T'}(x) - a.\mathit{freq} \cdot d_{T'}(a) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_T(a) - a.\mathit{freq} \cdot d_T(x) \\
&= (a.\mathit{freq} - x.\mathit{freq})(d_T(a) - d_T(x)) \\
&\geq 0 ,
\end{aligned}$$

because both $a.\mathit{freq} - x.\mathit{freq}$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.\mathit{freq} - x.\mathit{freq}$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T. Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Lemma 16.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 16.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix codes has the optimal-substructure property.

Lemma 16.3
Let C be a given alphabet with frequency $c.\mathit{freq}$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T', by considering the component costs in equation (16.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.\mathit{freq} \cdot d_T(c) = c.\mathit{freq} \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.\mathit{freq} \cdot d_T(x) + y.\mathit{freq} \cdot d_T(y) &= (x.\mathit{freq} + y.\mathit{freq})(d_{T'}(z) + 1) \\ &= z.\mathit{freq} \cdot d_{T'}(z) + (x.\mathit{freq} + y.\mathit{freq}) , \end{aligned}$$

from which we conclude that

$$B(T) = B(T') + x.\mathit{freq} + y.\mathit{freq}$$

or, equivalently,

$$B(T') = B(T) - x.\mathit{freq} - y.\mathit{freq} .$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 16.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.\mathit{freq} - y.\mathit{freq} \\ &< B(T) - x.\mathit{freq} - y.\mathit{freq} \\ &= B(T') , \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix code for C'. Thus, T must represent an optimal prefix code for the alphabet C. ■

Theorem 16.4
Procedure HUFFMAN produces an optimal prefix code.

Proof Immediate from Lemmas 16.2 and 16.3. ■

Exercises

16.3-1
Explain why, in the proof of Lemma 16.2, if $x.\mathit{freq} = b.\mathit{freq}$, then we must have $a.\mathit{freq} = b.\mathit{freq} = x.\mathit{freq} = y.\mathit{freq}$.

16.3-2
Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

16.3-3
What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

`a`:1 `b`:1 `c`:2 `d`:3 `e`:5 `f`:8 `g`:13 `h`:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

16.3-4
Prove that we can also express the total cost of a tree for a code as the sum, over all internal nodes, of the combined frequencies of the two children of the node.

16.3-5
Prove that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

16.3-6
Suppose we have an optimal prefix code on a set $C = \{0, 1, \ldots, n-1\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n - 1 + n\lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

16.3-7
Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

16.3-8
Suppose that a data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

16.3-9
Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit. (*Hint:* Compare the number of possible files with the number of possible encoded files.)

⋆ 16.4 Matroids and greedy methods

In this section, we sketch a beautiful theory about greedy algorithms. This theory describes many situations in which the greedy method yields optimal solutions. It involves combinatorial structures known as "matroids." Although this theory does not cover all cases for which a greedy method applies (for example, it does not cover the activity-selection problem of Section 16.1 or the Huffman-coding problem of Section 16.3), it does cover many cases of practical interest. Furthermore, this theory has been extended to cover many applications; see the notes at the end of this chapter for references.

Matroids

A ***matroid*** is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions.

1. S is a finite set.
2. $\mathcal{I}$ is a nonempty family of subsets of S, called the ***independent*** subsets of S, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. We say that $\mathcal{I}$ is ***hereditary*** if it satisfies this property. Note that the empty set $\emptyset$ is necessarily a member of $\mathcal{I}$.
3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there exists some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the ***exchange property***.

The word "matroid" is due to Hassler Whitney. He was studying ***matric matroids***, in which the elements of S are the rows of a given matrix and a set of rows is independent if they are linearly independent in the usual sense. As Exercise 16.4-2 asks you to show, this structure defines a matroid.

As another example of matroids, consider the ***graphic matroid*** $M_G = (S_G, \mathcal{I}_G)$ defined in terms of a given undirected graph $G = (V, E)$ as follows:

- The set S_G is defined to be E, the set of edges of G.
- If A is a subset of E, then $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

The graphic matroid M_G is closely related to the minimum-spanning-tree problem, which Chapter 23 covers in detail.

Amortized Analysis

17 Amortized Analysis

In an ***amortized analysis***, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 17.1 starts with aggregate analysis, in which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 17.2 covers the accounting method, in which we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 17.3 discusses the potential method, which is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.

We shall use two examples to examine these three methods. One is a stack with the additional operation MULTIPOP, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation INCREMENT.

While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They need not—and should not—appear in the code. If, for example, we assign a credit to an object x when using the accounting method, we have no need to assign an appropriate amount to some attribute, such as $x.credit$, in the code.

When we perform an amortized analysis, we often gain insight into a particular data structure, and this insight can help us optimize the design. In Section 17.4, for example, we shall use the potential method to analyze a dynamically expanding and contracting table.

17.1 Aggregate analysis

In ***aggregate analysis***, we show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or ***amortized cost***, per operation is therefore $T(n)/n$. Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

In our first example of aggregate analysis, we analyze stacks that have been augmented with a new operation. Section 10.1 presented the two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S.

POP(S) pops the top of stack S and returns the popped object. Calling POP on an empty stack generates an error.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n, and the actual running time for n operations is therefore $\Theta(n)$.

Now we add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. Of course, we assume that k is positive; otherwise the MULTIPOP operation leaves the stack unchanged. In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

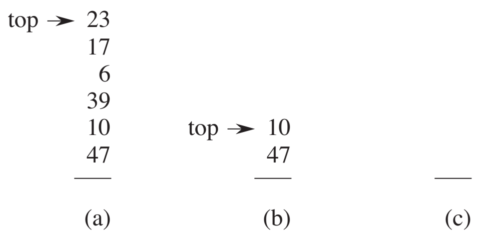


Figure 17.1 The action of MULTIPOP on a stack S, shown initially in **(a)**. The top 4 objects are popped by MULTIPOP$(S, 4)$, whose result is shown in **(b)**. The next operation is MULTIPOP$(S, 7)$, which empties the stack—shown in **(c)**—since there were fewer than 7 objects remaining.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k - 1
```

Figure 17.1 shows an example of MULTIPOP.

What is the running time of MULTIPOP(S, k) on a stack of s objects? The actual running time is linear in the number of POP operations actually executed, and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP. The number of iterations of the **while** loop is the number $\min(s, k)$ of objects popped off the stack. Each iteration of the loop makes one call to POP in line 2. Thus, the total cost of MULTIPOP is $\min(s, k)$, and the actual running time is a linear function of this cost.

Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, which we obtained by considering the worst-case cost of each operation individually, is not tight.

Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations. In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$. Why? We can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n. For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. The average cost of an operation is $O(n)/n = O(1)$. In aggregate analysis, we assign the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

We emphasize again that although we have just shown that the average cost, and hence the running time, of a stack operation is $O(1)$, we did not use probabilistic reasoning. We actually showed a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

Incrementing a binary counter

As another example of aggregate analysis, consider the problem of implementing a k-bit binary counter that counts upward from 0. We use an array $A[0\,..\,k-1]$ of bits, where $A.\mathit{length} = k$, as the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$. To add 1 (modulo 2^k) to the value in the counter, we use the following procedure.

```
INCREMENT($A$)
1  $i = 0$
2  while $i < A.length$ and $A[i] == 1$
3      $A[i] = 0$
4      $i = i + 1$
5  if $i < A.length$
6      $A[i] = 1$
```

Figure 17.2 shows what happens to a binary counter as we increment it 16 times, starting with the initial value 0 and ending with the value 16. At the start of each iteration of the **while** loop in lines 2–4, we wish to add a 1 into position i. If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, we know that $A[i] = 0$, so that line 6 adds a 1 into position i, flipping the 0 to a 1. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes time $\Theta(k)$ in the worst case, in which array A contains all 1s. Thus, a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.

We can tighten our analysis to yield a worst-case cost of $O(n)$ for a sequence of n INCREMENT operations by observing that not all bits flip each time INCREMENT is called. As Figure 17.2 shows, $A[0]$ does flip each time INCREMENT is called. The next bit up, $A[1]$, flips only every other time: a sequence of n INCREMENT

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT operations.

operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT operations. In general, for $i = 0, 1, \ldots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter. For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip. The total number of flips in the sequence is thus

$$\begin{aligned}\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n ,\end{aligned}$$

by equation (A.6). The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Exercises

17.1-1

If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

17.1-2

Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as $\Theta(nk)$ time.

17.1-3

Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

17.2 The accounting method

In the ***accounting method*** of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. We call the amount we charge an operation its ***amortized cost***. When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as ***credit***. Credit can help pay for later operations whose amortized cost is less than their actual cost. Thus, we can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. Different operations may have different amortized costs. This method differs from aggregate analysis, in which all operations have the same amortized cost.

We must choose the amortized costs of operations carefully. If we want to show that in the worst case the average cost per operation is small by analyzing with amortized costs, we must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, this relationship must hold for all sequences of operations. If we denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\widehat{c}_i$, we require

$$\sum_{i=1}^{n} \widehat{c}_i \ge \sum_{i=1}^{n} c_i \tag{17.1}$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or

$\sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i$. By inequality (17.1), the total credit associated with the data structure must be nonnegative at all times. If we ever were to allow the total credit to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack example. Recall that the actual costs of the operations were

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min(k, s)$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

Note that the amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. Recall the analogy of Section 10.1 between the stack data structure and a stack of plates in a cafeteria. When we push a plate on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we leave on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it.

The dollar stored on the plate serves as prepayment for the cost of popping it from the stack. When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To pop a plate, we take the dollar of credit off the plate and use it to pay the actual cost of the operation. Thus, by charging the PUSH operation a little bit more, we can charge the POP operation nothing.

Moreover, we can also charge MULTIPOP operations nothing. To pop the first plate, we take the dollar of credit off the plate and use it to pay the actual cost of a POP operation. To pop a second plate, we again have a dollar of credit on the plate to pay for the POP operation, and so on. Thus, we have always charged enough up front to pay for MULTIPOP operations. In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

Incrementing a binary counter

As another illustration of the accounting method, we analyze the INCREMENT operation on a binary counter that starts at zero. As we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we shall use as our cost for this example. Let us once again use a dollar bill to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, let us charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we can charge nothing to reset a bit to 0; we just pay for the reset with the dollar bill on the bit.

Now we can determine the amortized cost of INCREMENT. The cost of resetting the bits within the **while** loop is paid for by the dollars on the bits that are reset. The INCREMENT procedure sets at most one bit, in line 6, and therefore the amortized cost of an INCREMENT operation is at most 2 dollars. The number of 1s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

Exercises

17.2-1

Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

17.2-2

Redo Exercise 17.1-3 using an accounting method of analysis.

17.2-3

Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter. (*Hint:* Keep a pointer to the high-order 1.)

17.3 The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the ***potential method*** of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure.

The potential method works as follows. We will perform n operations, starting with an initial data structure D_0. For each $i = 1, 2, \ldots, n$, we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A ***potential function*** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the ***potential*** associated with data structure D_i. The ***amortized cost*** $\widehat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \tag{17.2}$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. By equation (17.2), the total amortized cost of the n operations is

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .\end{aligned} \tag{17.3}$$

The second equality follows from equation (A.9) because the $\Phi(D_i)$ terms telescope.

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \widehat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

In practice, we do not always know how many operations might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then we guarantee, as in the accounting method, that we pay in advance. We usually just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all i. (See Exercise 17.3-1 for an easy way to handle cases in which $\Phi(D_0) \neq 0$.)

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the ith operation is positive, then the amortized cost $\widehat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the ith operation, and the decrease in the potential pays for the actual cost of the operation.

The amortized costs defined by equations (17.2) and (17.3) depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs. We often find trade-offs that we can make in choosing a potential function; the best potential function to use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to the example of the stack operations PUSH, POP, and MULTIPOP. We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack D_0 with which we start, we have $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0) . \end{aligned}$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Let us now compute the amortized costs of the various stack operations. If the ith operation on a stack containing s objects is a PUSH operation, then the potential difference is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 . \end{aligned}$$

By equation (17.2), the amortized cost of this PUSH operation is

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 . \end{aligned}$$

Suppose that the ith operation on the stack is MULTIPOP(S, k), which causes $k' = \min(k, s)$ objects to be popped off the stack. The actual cost of the operation is k', and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' .$$

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 . \end{aligned}$$

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Incrementing a binary counter

As another example of the potential method, we again look at incrementing a binary counter. This time, we define the potential of the counter after the ith INCREMENT operation to be b_i, the number of 1s in the counter after the ith operation.

Let us compute the amortized cost of an INCREMENT operation. Suppose that the ith INCREMENT operation resets t_i bits. The actual cost of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1. If $b_i = 0$, then the ith operation resets all k bits, and so $b_{i-1} = t_i = k$. If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$. In either case, $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i . \end{aligned}$$

The amortized cost is therefore

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 . \end{aligned}$$

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method gives us an easy way to analyze the counter even when it does not start at zero. The counter starts with b_0 1s, and after n INCREMENT operations it has b_n 1s, where $0 \le b_0, b_n \le k$. (Recall that k is the number of bits in the counter.) We can rewrite equation (17.3) as

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) . \tag{17.4}$$

We have $\hat{c}_i \le 2$ for all $1 \le i \le n$. Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 .\end{aligned}$$

Note in particular that since $b_0 \le k$, as long as $k = O(n)$, the total actual cost is $O(n)$. In other words, if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Exercises

17.3-1
Suppose we have a potential function Φ such that $\Phi(D_i) \ge \Phi(D_0)$ for all i, but $\Phi(D_0) \ne 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \ge 0$ for all $i \ge 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

17.3-2
Redo Exercise 17.1-3 using a potential method of analysis.

17.3-3
Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.

17.3-4
What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

17.3-5
Suppose that a counter begins at a number with b 1s in its binary representation, rather than at 0. Show that the cost of performing n INCREMENT operations is $O(n)$ if $n = \Omega(b)$. (Do not assume that b is constant.)

17.3-6
Show how to implement a queue with two ordinary stacks (Exercise 10.1-6) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

17.3-7
Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT(S, x) inserts x into S.

DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

17.4 Dynamic tables

We do not always know in advance how many objects some applications will store in a table. We might allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table. Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size. In this section, we study this problem of dynamically expanding and contracting a table. Using amortized analysis, we shall show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, we shall see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a single ***slot***, that is, a space for one item. Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot. The details of the data-structuring method used to organize the table are unimportant; we might use a stack (Section 10.1), a heap (Chapter 6), or a hash table (Chapter 11). We might also use an array or collection of arrays to implement object storage, as we did in Section 10.3.

We shall find it convenient to use a concept introduced in our analysis of hashing (Chapter 11). We define the ***load factor*** $\alpha(T)$ of a nonempty table T to be the number of items stored in the table divided by the size (number of slots) of the table. We assign an empty table (one with no items) size 0, and we define its load factor to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.

We start by analyzing a dynamic table in which we only insert items. We then consider the more general case in which we both insert and delete items.

17.4.1 Table expansion

Let us assume that storage for a table is allocated as an array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1.[1] In some software environments, upon attempting to insert an item into a full table, the only alternative is to abort with an error. We shall assume, however, that our software environment, like many modern ones, provides a memory-management system that can allocate and free blocks of storage on request. Thus, upon inserting an item into a full table, we can ***expand*** the table by allocating a new table with more slots than the old table had. Because we always need the table to reside in contiguous memory, we must allocate a new array for the larger table and then copy items from the old table into the new table.

A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the load factor of the table is always at least $1/2$, and thus the amount of wasted space never exceeds half the total space in the table.

In the following pseudocode, we assume that T is an object representing the table. The attribute $T.table$ contains a pointer to the block of storage representing the table, $T.num$ contains the number of items in the table, and $T.size$ gives the total number of slots in the table. Initially, the table is empty: $T.num = T.size = 0$.

```
TABLE-INSERT(T, x)
 1  if $T.size == 0$
 2      allocate $T.table$ with 1 slot
 3      $T.size = 1$
 4  if $T.num == T.size$
 5      allocate $new\text{-}table$ with $2 \cdot T.size$ slots
 6      insert all items in $T.table$ into $new\text{-}table$
 7      free $T.table$
 8      $T.table = new\text{-}table$
 9      $T.size = 2 \cdot T.size$
10  insert $x$ into $T.table$
11  $T.num = T.num + 1$
```

[1] In some situations, such as an open-address hash table, we may wish to consider a table to be full if its load factor equals some constant strictly less than 1. (See Exercise 17.4-1.)

Notice that we have two "insertion" procedures here: the TABLE-INSERT procedure itself and the ***elementary insertion*** into a table in lines 6 and 10. We can analyze the running time of TABLE-INSERT in terms of the number of elementary insertions by assigning a cost of 1 to each elementary insertion. We assume that the actual running time of TABLE-INSERT is linear in the time to insert individual items, so that the overhead for allocating an initial table in line 2 is constant and the overhead for allocating and freeing storage in lines 5 and 7 is dominated by the cost of transferring items in line 6. We call the event in which lines 5–9 are executed an ***expansion***.

Let us analyze a sequence of n TABLE-INSERT operations on an initially empty table. What is the cost c_i of the ith operation? If the current table has room for the new item (or if this is the first operation), then $c_i = 1$, since we need only perform the one elementary insertion in line 10. If the current table is full, however, and an expansion occurs, then $c_i = i$: the cost is 1 for the elementary insertion in line 10 plus $i - 1$ for the items that we must copy from the old table to the new table in line 6. If we perform n operations, the worst-case cost of an operation is $O(n)$, which leads to an upper bound of $O(n^2)$ on the total running time for n operations.

This bound is not tight, because we rarely expand the table in the course of n TABLE-INSERT operations. Specifically, the ith operation causes an expansion only when $i - 1$ is an exact power of 2. The amortized cost of an operation is in fact $O(1)$, as we can show using aggregate analysis. The cost of the ith operation is

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2}, \\ 1 & \text{otherwise}. \end{cases}$$

The total cost of n TABLE-INSERT operations is therefore

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n , \end{aligned}$$

because at most n operations cost 1 and the costs of the remaining operations form a geometric series. Since the total cost of n TABLE-INSERT operations is bounded by $3n$, the amortized cost of a single operation is at most 3.

By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3. Intuitively, each item pays for 3 elementary insertions: inserting itself into the current table, moving itself when the table expands, and moving another item that has already been moved once when the table expands. For example, suppose that the size of the table is m immediately after an expansion. Then the table holds $m/2$ items, and it contains no credit. We charge 3 dollars for each insertion. The elementary insertion that occurs immediately costs 1 dollar. We place another dollar as credit on the item inserted. We place the third dollar as credit on one of the $m/2$ items already in the table. The table will not fill again until we have inserted another $m/2 - 1$ items, and thus, by the time the table contains m items and is full, we will have placed a dollar on each item to pay to reinsert it during the expansion.

We can use the potential method to analyze a sequence of n TABLE-INSERT operations, and we shall use it in Section 17.4.2 to design a TABLE-DELETE operation that has an $O(1)$ amortized cost as well. We start by defining a potential function Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full, so that we can pay for the next expansion by the potential. The function

$$\Phi(T) = 2 \cdot T.num - T.size \tag{17.5}$$

is one possibility. Immediately after an expansion, we have $T.num = T.size/2$, and thus $\Phi(T) = 0$, as desired. Immediately before an expansion, we have $T.num = T.size$, and thus $\Phi(T) = T.num$, as desired. The initial value of the potential is 0, and since the table is always at least half full, $T.num \geq T.size/2$, which implies that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.

To analyze the amortized cost of the ith TABLE-INSERT operation, we let num_i denote the number of items stored in the table after the ith operation, $size_i$ denote the total size of the table after the ith operation, and Φ_i denote the potential after the ith operation. Initially, we have $num_0 = 0$, $size_0 = 0$, and $\Phi_0 = 0$.

If the ith TABLE-INSERT operation does not trigger an expansion, then we have $size_i = size_{i-1}$ and the amortized cost of the operation is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 3 .
\end{aligned}$$

If the ith operation does trigger an expansion, then we have $size_i = 2 \cdot size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, which implies that $size_i = 2 \cdot (num_i - 1)$. Thus, the amortized cost of the operation is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3 .
\end{aligned}$$

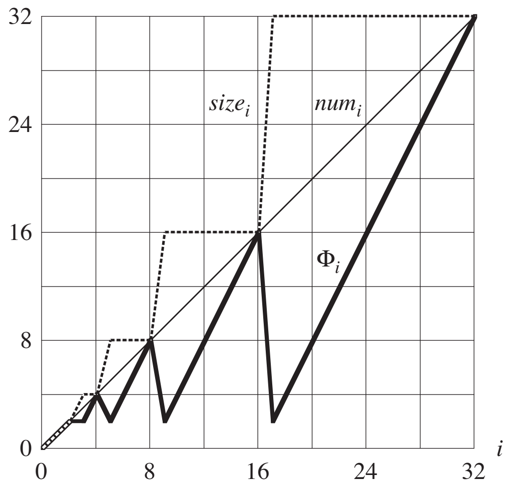


Figure 17.3 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential $\Phi_i = 2 \cdot num_i - size_i$, each being measured after the ith operation. The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i. Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterwards, the potential drops to 0, but it is immediately increased by 2 upon inserting the item that caused the expansion.

Figure 17.3 plots the values of num_i, $size_i$, and Φ_i against i. Notice how the potential builds to pay for expanding the table.

17.4.2 Table expansion and contraction

To implement a TABLE-DELETE operation, it is simple enough to remove the specified item from the table. In order to limit the amount of wasted space, however, we might wish to ***contract*** the table when the load factor becomes too small. Table contraction is analogous to table expansion: when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one. We can then free the storage for the old table by returning it to the memory-management system. Ideally, we would like to preserve two properties:

- the load factor of the dynamic table is bounded below by a positive constant, and
- the amortized cost of a table operation is bounded above by a constant.

We assume that we measure the cost in terms of elementary insertions and deletions.

You might think that we should double the table size upon inserting an item into a full table and halve the size when a deleting an item would cause the table to become less than half full. This strategy would guarantee that the load factor of the table never drops below $1/2$, but unfortunately, it can cause the amortized cost of an operation to be quite large. Consider the following scenario. We perform n operations on a table T, where n is an exact power of 2. The first $n/2$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$. At the end of this sequence of insertions, $T.num = T.size = n/2$. For the second $n/2$ operations, we perform the following sequence:

insert, delete, delete, insert, insert, delete, delete, insert, insert, ...

The first insertion causes the table to expand to size n. The two following deletions cause the table to contract back to size $n/2$. Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is $\Theta(n)$, and there are $\Theta(n)$ of them. Thus, the total cost of the n operations is $\Theta(n^2)$, making the amortized cost of an operation $\Theta(n)$.

The downside of this strategy is obvious: after expanding the table, we do not delete enough items to pay for a contraction. Likewise, after contracting the table, we do not insert enough items to pay for an expansion.

We can improve upon this strategy by allowing the load factor of the table to drop below $1/2$. Specifically, we continue to double the table size upon inserting an item into a full table, but we halve the table size when deleting an item causes the table to become less than $1/4$ full, rather than $1/2$ full as before. The load factor of the table is therefore bounded below by the constant $1/4$.

Intuitively, we would consider a load factor of $1/2$ to be ideal, and the table's potential would then be 0. As the load factor deviates from $1/2$, the potential increases so that by the time we expand or contract the table, the table has garnered sufficient potential to pay for copying all the items into the newly allocated table. Thus, we will need a potential function that has grown to $T.num$ by the time that the load factor has either increased to 1 or decreased to $1/4$. After either expanding or contracting the table, the load factor goes back to $1/2$ and the table's potential reduces back to 0.

We omit the code for TABLE-DELETE, since it is analogous to TABLE-INSERT. For our analysis, we shall assume that whenever the number of items in the table drops to 0, we free the storage for the table. That is, if $T.num = 0$, then $T.size = 0$.

We can now use the potential method to analyze the cost of a sequence of n TABLE-INSERT and TABLE-DELETE operations. We start by defining a potential function Φ that is 0 immediately after an expansion or contraction and builds as the load factor increases to 1 or decreases to $1/4$. Let us denote the load fac-

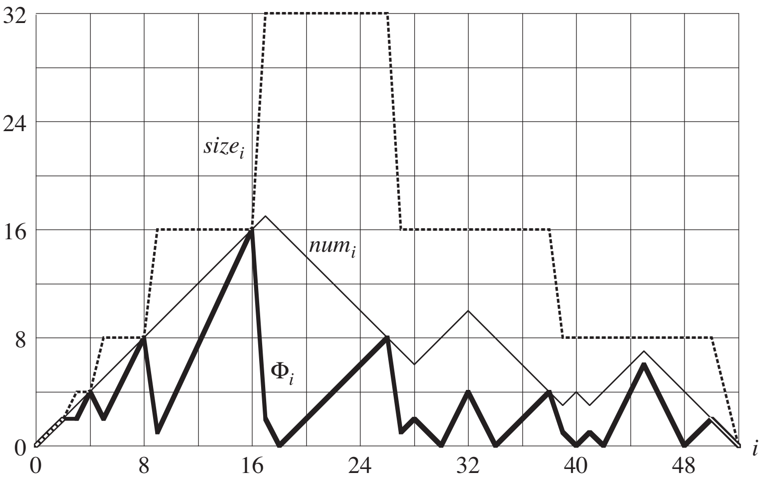


Figure 17.4 The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{if } \alpha_i \geq 1/2 , \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2 , \end{cases}$$

each measured after the ith operation. The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i. Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Likewise, immediately before a contraction, the potential has built up to the number of items in the table.

tor of a nonempty table T by $\alpha(T) = T.num/T.size$. Since for an empty table, $T.num = T.size = 0$ and $\alpha(T) = 1$, we always have $T.num = \alpha(T) \cdot T.size$, whether the table is empty or not. We shall use as our potential function

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{if } \alpha(T) \geq 1/2 , \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 . \end{cases} \tag{17.6}$$

Observe that the potential of an empty table is 0 and that the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence.

Before proceeding with a precise analysis, we pause to observe some properties of the potential function, as illustrated in Figure 17.4. Notice that when the load factor is $1/2$, the potential is 0. When the load factor is 1, we have $T.size = T.num$, which implies $\Phi(T) = T.num$, and thus the potential can pay for an expansion if an item is inserted. When the load factor is $1/4$, we have $T.size = 4 \cdot T.num$, which implies $\Phi(T) = T.num$, and thus the potential can pay for a contraction if an item is deleted.

To analyze a sequence of n TABLE-INSERT and TABLE-DELETE operations, we let c_i denote the actual cost of the ith operation, $\widehat{c}_i$ denote its amortized cost with respect to Φ, num_i denote the number of items stored in the table after the ith operation, $size_i$ denote the total size of the table after the ith operation, α_i denote the load factor of the table after the ith operation, and Φ_i denote the potential after the ith operation. Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$, and $\Phi_0 = 0$.

We start with the case in which the ith operation is TABLE-INSERT. The analysis is identical to that for table expansion in Section 17.4.1 if $\alpha_{i-1} \geq 1/2$. Whether the table expands or not, the amortized cost $\widehat{c}_i$ of the operation is at most 3. If $\alpha_{i-1} < 1/2$, the table cannot expand as a result of the operation, since the table expands only when $\alpha_{i-1} = 1$. If $\alpha_i < 1/2$ as well, then the amortized cost of the ith operation is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\
&= 0 .
\end{aligned}$$

If $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$, then

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3 .
\end{aligned}$$

Thus, the amortized cost of a TABLE-INSERT operation is at most 3.

We now turn to the case in which the ith operation is TABLE-DELETE. In this case, $num_i = num_{i-1} - 1$. If $\alpha_{i-1} < 1/2$, then we must consider whether the operation causes the table to contract. If it does not, then $size_i = size_{i-1}$ and the amortized cost of the operation is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\
&= 2 .
\end{aligned}$$

If $\alpha_{i-1} < 1/2$ and the ith operation does trigger a contraction, then the actual cost of the operation is $c_i = num_i + 1$, since we delete one item and move num_i items. We have $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$, and the amortized cost of the operation is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\
&= 1 .
\end{aligned}$$

When the ith operation is a TABLE-DELETE and $\alpha_{i-1} \geq 1/2$, the amortized cost is also bounded above by a constant. We leave the analysis as Exercise 17.4-2.

In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of n operations on a dynamic table is $O(n)$.

Exercises

17.4-1

Suppose that we wish to implement a dynamic, open-address hash table. Why might we consider the table to be full when its load factor reaches some value α that is strictly less than 1? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

17.4-2

Show that if $\alpha_{i-1} \geq 1/2$ and the ith operation on a dynamic table is TABLE-DELETE, then the amortized cost of the operation with respect to the potential function (17.6) is bounded above by a constant.

17.4-3

Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, we contract it by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function

$$\Phi(T) = |2 \cdot T.num - T.size| ,$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

Problems

17-1 Bit-reversed binary counter

Chapter 30 examines an important algorithm called the fast Fourier transform, or FFT. The first step of the FFT algorithm performs a ***bit-reversal permutation*** on an input array $A[0 \mathrel{.\,.} n-1]$ whose length is $n = 2^k$ for some nonnegative integer k. This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index a as a k-bit sequence $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$, where $a = \sum_{i=0}^{k-1} a_i\, 2^i$. We define

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1, \ldots, a_{k-1} \rangle ;$$

thus,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i .$$

For example, if $n = 16$ (or, equivalently, $k = 4$), then $\text{rev}_k(3) = 12$, since the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit representation of 12.

a. Given a function rev_k that runs in $\Theta(k)$ time, write an algorithm to perform the bit-reversal permutation on an array of length $n = 2^k$ in $O(nk)$ time.

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a "bit-reversed counter" and a procedure BIT-REVERSED-INCREMENT that, when given a bit-reversed-counter value a, produces $\text{rev}_k(\text{rev}_k(a) + 1)$. If $k = 4$, for example, and the bit-reversed counter starts at 0, then successive calls to BIT-REVERSED-INCREMENT produce the sequence

$$0000, 1000, 0100, 1100, 0010, 1010, \ldots = 0, 8, 4, 12, 2, 10, \ldots .$$

b. Assume that the words in your computer store k-bit values and that in unit time, your computer can manipulate the binary values with operations such as shifting left or right by arbitrary amounts, bitwise-AND, bitwise-OR, etc. Describe an implementation of the BIT-REVERSED-INCREMENT procedure that allows the bit-reversal permutation on an n-element array to be performed in a total of $O(n)$ time.

c. Suppose that you can shift a word left or right by only one bit in unit time. Is it still possible to implement an $O(n)$-time bit-reversal permutation?

17-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We have k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i\, 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.

c. Discuss how to implement DELETE.

17-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$ giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \le \alpha < 1$. We say that a given node x is ***α-balanced*** if $x.left.size \le \alpha \cdot x.size$ and $x.right.size \le \alpha \cdot x.size$. The tree as a whole is ***α-balanced*** if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A $1/2$-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $1/2$-balanced. Your algorithm should run in time $\Theta(x.size)$, and it can use $O(x.size)$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than $1/2$. Suppose that we implement INSERT and DELETE as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then we "rebuild" the subtree rooted at the highest such node in the tree so that it becomes $1/2$-balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, we define

$$\Delta(x) = |x.\mathit{left.size} - x.\mathit{right.size}| ,$$

and we define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x) ,$$

where c is a sufficiently large constant that depends on α.

c. Argue that any binary search tree has nonnegative potential and that a 1/2-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.

17-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform ***structural modifications***: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

a. Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n + 1)$st node causes $\Omega(\lg n)$ color changes. Then describe a legal red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color changes.

Although the worst-case number of color changes per operation can be logarithmic, we shall prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case. Note that we count each color change as a structural modification.

b. Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are ***terminating***: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (*Hint:* Look at Figures 13.5, 13.6, and 13.7.)

We shall first analyze the structural modifications when only insertions are performed. Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T. Assume that 1 unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

c. Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T. Argue that $\Phi(T') = \Phi(T) - 1$.

d. When we insert a node into a red-black tree using RB-INSERT, we can break the operation into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

We now wish to prove that there are $O(m)$ structural modifications when there are both insertions and deletions. Let us define, for each node x,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red} , \\ 1 & \text{if } x \text{ is black and has no red children} , \\ 0 & \text{if } x \text{ is black and has one red child} , \\ 2 & \text{if } x \text{ is black and has two red children} . \end{cases}$$

Now we redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x) ,$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T.

f. Show that $\Phi(T') \le \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

g. Show that $\Phi(T') \le \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

h. Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.

17-5 Competitive analysis of self-organizing lists with move-to-front

A ***self-organizing list*** is a linked list of n elements, in which each element has a unique key. When we search for an element in the list, we are given a key, and we want to find an element with that key.

A self-organizing list has two important properties:

1. To find an element in the list, given its key, we must traverse the list from the beginning until we encounter the element with the given key. If that element is the kth element from the start of the list, then the cost to find the element is k.
2. We may reorder the list elements after any operation, according to a given rule with a given cost. We may choose any heuristic we like to decide how to reorder the list.

Assume that we start with a given list of n elements, and we are given an access sequence $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_m \rangle$ of keys to find, in order. The cost of the sequence is the sum of the costs of the individual accesses in the sequence.

Out of the various possible ways to reorder the list after an operation, this problem focuses on transposing adjacent list elements—switching their positions in the list—with a unit cost for each transpose operation. You will show, by means of a potential function, that a particular heuristic for reordering the list, move-to-front, entails a total cost no worse than 4 times that of any other heuristic for maintaining the list order—even if the other heuristic knows the access sequence in advance! We call this type of analysis a ***competitive analysis***.

For a heuristic H and a given initial ordering of the list, denote the access cost of sequence σ by $C_H(\sigma)$. Let m be the number of accesses in σ.

a. Argue that if heuristic H does not know the access sequence in advance, then the worst-case cost for H on an access sequence σ is $C_H(\sigma) = \Omega(mn)$.

With the ***move-to-front*** heuristic, immediately after searching for an element x, we move x to the first position on the list (i.e., the front of the list).

Let $\text{rank}_L(x)$ denote the rank of element x in list L, that is, the position of x in list L. For example, if x is the fourth element in L, then $\text{rank}_L(x) = 4$. Let c_i denote the cost of access σ_i using the move-to-front heuristic, which includes the cost of finding the element in the list and the cost of moving it to the front of the list by a series of transpositions of adjacent list elements.

b. Show that if σ_i accesses element x in list L using the move-to-front heuristic, then $c_i = 2 \cdot \text{rank}_L(x) - 1$.

Now we compare move-to-front with any other heuristic H that processes an access sequence according to the two properties above. Heuristic H may transpose elements in the list in any way it wants, and it might even know the entire access sequence in advance.

Let L_i be the list after access σ_i using move-to-front, and let L_i^* be the list after access σ_i using heuristic H. We denote the cost of access σ_i by c_i for move-to-front and by c_i^* for heuristic H. Suppose that heuristic H performs t_i^* transpositions during access σ_i.

c. In part (b), you showed that $c_i = 2 \cdot \text{rank}_{L_{i-1}}(x) - 1$. Now show that $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i^*$.

We define an ***inversion*** in list L_i as a pair of elements y and z such that y precedes z in L_i and z precedes y in list L_i^*. Suppose that list L_i has q_i inversions after processing the access sequence $\langle \sigma_1, \sigma_2, \ldots, \sigma_i \rangle$. Then, we define a potential function Φ that maps L_i to a real number by $\Phi(L_i) = 2q_i$. For example, if L_i has the elements $\langle e, c, a, d, b \rangle$ and L_i^* has the elements $\langle c, a, b, d, e \rangle$, then L_i has 5 inversions ($(e,c), (e,a), (e,d), (e,b), (d,b)$), and so $\Phi(L_i) = 10$. Observe that $\Phi(L_i) \geq 0$ for all i and that, if move-to-front and heuristic H start with the same list L_0, then $\Phi(L_0) = 0$.

d. Argue that a transposition either increases the potential by 2 or decreases the potential by 2.

Suppose that access σ_i finds the element x. To understand how the potential changes due to σ_i, let us partition the elements other than x into four sets, depending on where they are in the lists just before the ith access:

- Set A consists of elements that precede x in both L_{i-1} and L_{i-1}^*.
- Set B consists of elements that precede x in L_{i-1} and follow x in L_{i-1}^*.
- Set C consists of elements that follow x in L_{i-1} and precede x in L_{i-1}^*.
- Set D consists of elements that follow x in both L_{i-1} and L_{i-1}^*.

e. Argue that $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ and $\text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$.

f. Show that access σ_i causes a change in potential of

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i^*) ,$$

where, as before, heuristic H performs t_i^* transpositions during access σ_i.

Define the amortized cost $\widehat{c}_i$ of access σ_i by $\widehat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$.

g. Show that the amortized cost $\widehat{c}_i$ of access σ_i is bounded from above by $4c_i^*$.

h. Conclude that the cost $C_{\text{MTF}}(\sigma)$ of access sequence σ with move-to-front is at most 4 times the cost $C_H(\sigma)$ of σ with any other heuristic H, assuming that both heuristics start with the same list.

Chapter notes

Aho, Hopcroft, and Ullman [5] used aggregate analysis to determine the running time of operations on a disjoint-set forest; we shall analyze this data structure using the potential method in Chapter 21. Tarjan [331] surveys the accounting and potential methods of amortized analysis and presents several applications. He attributes the accounting method to several authors, including M. R. Brown, R. E. Tarjan, S. Huddleston, and K. Mehlhorn. He attributes the potential method to D. D. Sleator. The term "amortized" is due to D. D. Sleator and R. E. Tarjan.

Potential functions are also useful for proving lower bounds for certain types of problems. For each configuration of the problem, we define a potential function that maps the configuration to a real number. Then we determine the potential Φ_{init} of the initial configuration, the potential Φ_{final} of the final configuration, and the maximum change in potential $\Delta\Phi_{\text{max}}$ due to any step. The number of steps must therefore be at least $|\Phi_{\text{final}} - \Phi_{\text{init}}| / |\Delta\Phi_{\text{max}}|$. Examples of potential functions to prove lower bounds in I/O complexity appear in works by Cormen, Sundquist, and Wisniewski [79]; Floyd [107]; and Aggarwal and Vitter [3]. Krumme, Cybenko, and Venkataraman [221] applied potential functions to prove lower bounds on ***gossiping***: communicating a unique item from each vertex in a graph to every other vertex.

The move-to-front heuristic from Problem 17-5 works quite well in practice. Moreover, if we recognize that when we find an element, we can splice it out of its position in the list and relocate it to the front of the list in constant time, we can show that the cost of move-to-front is at most twice the cost of any other heuristic including, again, one that knows the entire access sequence in advance.

B-Tree

18 B-Trees

B-trees are balanced search trees designed to work well on disks or other direct-access secondary storage devices. B-trees are similar to red-black trees (Chapter 13), but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a few to thousands. That is, the "branching factor" of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used. B-trees are similar to red-black trees in that every n-node B-tree has height $O(\lg n)$. The exact height of a B-tree can be considerably less than that of a red-black tree, however, because its branching factor, and hence the base of the logarithm that expresses its height, can be much larger. Therefore, we can also use B-trees to implement many dynamic-set operations in time $O(\lg n)$.

B-trees generalize binary search trees in a natural manner. Figure 18.1 shows a simple B-tree. If an internal B-tree node x contains $x.n$ keys, then x has $x.n + 1$ children. The keys in node x serve as dividing points separating the range of keys handled by x into $x.n + 1$ subranges, each handled by one child of x. When searching for a key in a B-tree, we make an $(x.n + 1)$-way decision based on comparisons with the $x.n$ keys stored at node x. The structure of leaf nodes differs from that of internal nodes; we will examine these differences in Section 18.1.

Section 18.1 gives a precise definition of B-trees and proves that the height of a B-tree grows only logarithmically with the number of nodes it contains. Section 18.2 describes how to search for a key and insert a key into a B-tree, and Section 18.3 discusses deletion. Before proceeding, however, we need to ask why we evaluate data structures designed to work on a disk differently from data structures designed to work in main random-access memory.

Data structures on secondary storage

Computer systems take advantage of various technologies that provide memory capacity. The ***primary memory*** (or ***main memory***) of a computer system normally

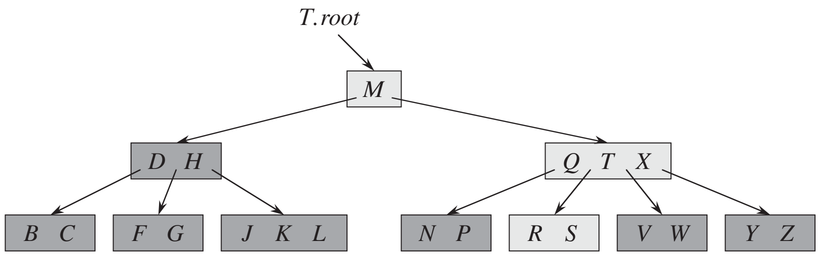


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

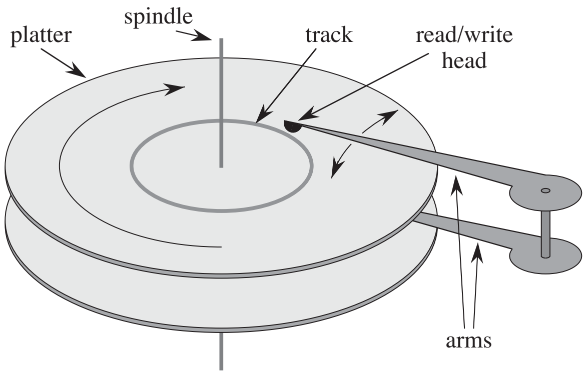


Figure 18.2 A typical disk drive. It comprises one or more platters (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head at the end of an arm. Arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when the head is stationary.

consists of silicon memory chips. This technology is typically more than an order of magnitude more expensive per bit stored than magnetic storage technology, such as tapes or disks. Most computer systems also have ***secondary storage*** based on magnetic disks; the amount of such secondary storage often exceeds the amount of primary memory by at least two orders of magnitude.

Figure 18.2 shows a typical disk drive. The drive consists of one or more ***platters***, which rotate at a constant speed around a common ***spindle***. A magnetizable material covers the surface of each platter. The drive reads and writes each platter by a ***head*** at the end of an ***arm***. The arms can move their heads toward or away from the spindle. When a given head is stationary, the surface that passes underneath it is called a ***track***. Multiple platters increase only the disk drive's capacity and not its performance.

Although disks are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts.[1] The mechanical motion has two components: platter rotation and arm movement. As of this writing, commodity disks rotate at speeds of 5400–15,000 revolutions per minute (RPM). We typically see 15,000 RPM speeds in server-grade drives, 7200 RPM speeds in drives for desktops, and 5400 RPM speeds in drives for laptops. Although 7200 RPM may seem fast, one rotation takes 8.33 milliseconds, which is over 5 orders of magnitude longer than the 50 nanosecond access times (more or less) commonly found for silicon memory. In other words, if we have to wait a full rotation for a particular item to come under the read/write head, we could access main memory more than 100,000 times during that span. On average we have to wait for only half a rotation, but still, the difference in access times for silicon memory compared with disks is enormous. Moving the arms also takes some time. As of this writing, average access times for commodity disks are in the range of 8 to 11 milliseconds.

In order to amortize the time spent waiting for mechanical movements, disks access not just one item but several at a time. Information is divided into a number of equal-sized ***pages*** of bits that appear consecutively within tracks, and each disk read or write is of one or more entire pages. For a typical disk, a page might be 2^{11} to 2^{14} bytes in length. Once the read/write head is positioned correctly and the disk has rotated to the beginning of the desired page, reading or writing a magnetic disk is entirely electronic (aside from the rotation of the disk), and the disk can quickly read or write large amounts of data.

Often, accessing a page of information and reading it from a disk takes longer than examining all the information read. For this reason, in this chapter we shall look separately at the two principal components of the running time:

- the number of disk accesses, and
- the CPU (computing) time.

We measure the number of disk accesses in terms of the number of pages of information that need to be read from or written to the disk. We note that disk-access time is not constant—it depends on the distance between the current track and the desired track and also on the initial rotational position of the disk. We shall

[1] As of this writing, solid-state drives have recently come onto the consumer market. Although they are faster than mechanical disk drives, they cost more per gigabyte and have lower capacities than mechanical disk drives.

nonetheless use the number of pages read or written as a first-order approximation of the total time spent accessing the disk.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed. B-tree algorithms keep only a constant number of pages in main memory at any time; thus, the size of main memory does not limit the size of B-trees that can be handled.

We model disk operations in our pseudocode as follows. Let x be a pointer to an object. If the object is currently in the computer's main memory, then we can refer to the attributes of the object as usual: $x.key$, for example. If the object referred to by x resides on disk, however, then we must perform the operation DISK-READ(x) to read object x into main memory before we can refer to its attributes. (We assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses; it is a "no-op.") Similarly, the operation DISK-WRITE(x) is used to save any changes that have been made to the attributes of object x. That is, the typical pattern for working with an object is as follows:

```
$x$ = a pointer to some object
DISK-READ($x$)
operations that access and/or modify the attributes of $x$
DISK-WRITE($x$)            // omitted if no attributes of $x$ were changed
other operations that access but do not modify attributes of $x$
```

The system can keep only a limited number of pages in main memory at any one time. We shall assume that the system flushes from main memory pages no longer in use; our B-tree algorithms will ignore this issue.

Since in most systems the running time of a B-tree algorithm depends primarily on the number of DISK-READ and DISK-WRITE operations it performs, we typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk page, and this size limits the number of children a B-tree node can have.

For a large B-tree stored on a disk, we often see branching factors between 50 and 2000, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key. Figure 18.3 shows a B-tree with a branching factor of 1001 and height 2 that can store over one billion keys; nevertheless, since we can keep the root node permanently in main memory, we can find any key in this tree by making at most only two disk accesses.

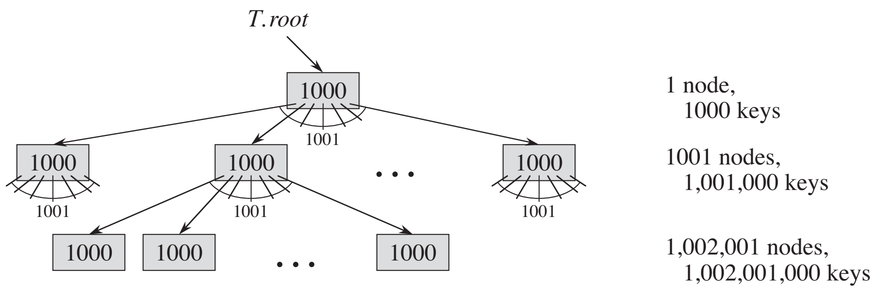

T.root
1000
1001
1 node, 1000 keys
1001 nodes, 1,001,000 keys
1,002,001 nodes, 1,002,001,000 keys

Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x is $x.n$, the number of keys in x. Each internal node and leaf contains 1000 keys. This B-tree has 1001 nodes at depth 1 and over one million leaves at depth 2.

18.1 Definition of B-trees

To keep things simple, we assume, as we have for binary search trees and red-black trees, that any "satellite information" associated with a key resides in the same node as the key. In practice, one might actually store with each key just a pointer to another disk page containing the satellite information for that key. The pseudocode in this chapter implicitly assumes that the satellite information associated with a key, or the pointer to such satellite information, travels with the key whenever the key is moved from node to node. A common variant on a B-tree, known as a ***B$^+$-tree***, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A ***B-tree*** T is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node x has the following attributes:

   a. $x.n$, the number of keys currently stored in node x,

   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \le x.key_2 \le \cdots \le x.key_{x.n}$,

   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains $x.n+1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \cdots \le x.key_{x.n} \le k_{x.n+1} .$$

4. All leaves have the same depth, which is the tree's height h.

5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:

   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[2]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. We now analyze the worst-case height of a B-tree.

Theorem 18.1
If $n \ge 1$, then for any n-key B-tree T of height h and minimum degree $t \ge 2$,

$$h \le \log_t \frac{n+1}{2} .$$

Proof The root of a B-tree T contains at least one key, and all other nodes contain at least $t - 1$ keys. Thus, T, whose height is h, has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h it has at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. Thus, the

[2]Another common variant on a B-tree, known as a ***B*-tree***, requires each internal node to be at least $2/3$ full, rather than at least half full, as a B-tree requires.

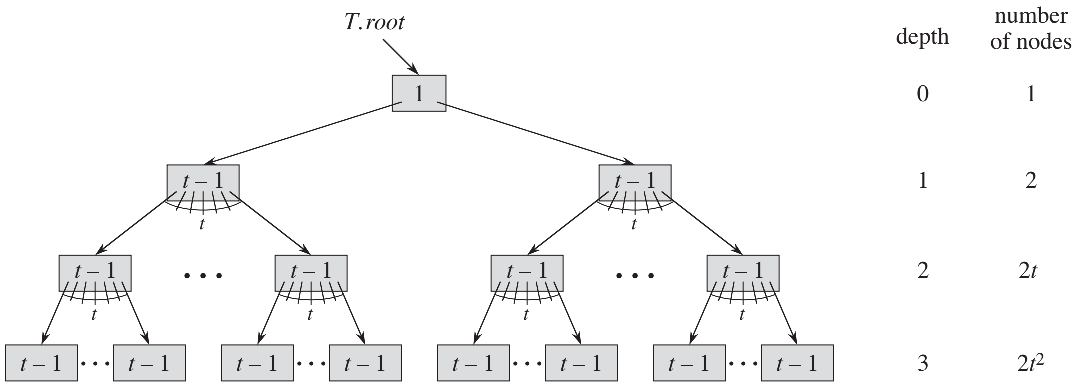


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.

number n of keys satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 \,. \end{aligned}$$

By simple algebra, we get $t^h \leq (n+1)/2$. Taking base-t logarithms of both sides proves the theorem. ■

Here we see the power of B-trees, as compared with red-black trees. Although the height of the tree grows as $O(\lg n)$ in both cases (recall that t is a constant), for B-trees the base of the logarithm can be many times larger. Thus, B-trees save a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Because we usually have to access the disk to examine an arbitrary node in a tree, B-trees avoid a substantial number of disk accesses.

Exercises

18.1-1
Why don't we allow a minimum degree of $t = 1$?

18.1-2
For what values of t is the tree of Figure 18.1 a legal B-tree?

18.1-3
Show all legal B-trees of minimum degree 2 that represent $\{1, 2, 3, 4, 5\}$.

18.1-4
As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

18.1-5
Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

18.2 Basic operations on B-trees

In this section, we present the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. In these procedures, we adopt two conventions:

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root; we do have to perform a DISK-WRITE of the root, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

The procedures we present are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

Searching a B-tree

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, we make a multiway branching decision according to the number of the node's children. More precisely, at each internal node x, we make an $(x.n + 1)$-way branching decision.

B-TREE-SEARCH is a straightforward generalization of the TREE-SEARCH procedure defined for binary search trees. B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH$(T.root, k)$. If k is in the B-tree, B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that $y.key_i = k$. Otherwise, the procedure returns NIL.

```
B-TREE-SEARCH$(x, k)$
1  $i = 1$
2  while $i \le x.n$ and $k > x.key_i$
3      $i = i + 1$
4  if $i \le x.n$ and $k == x.key_i$
5      return $(x, i)$
6  elseif $x.leaf$
7      return NIL
8  else DISK-READ$(x.c_i)$
9      return B-TREE-SEARCH$(x.c_i, k)$
```

Using a linear-search procedure, lines 1–3 find the smallest index i such that $k \le x.key_i$, or else they set i to $x.n + 1$. Lines 4–5 check to see whether we have now discovered the key, returning if we have. Otherwise, lines 6–9 either terminate the search unsuccessfully (if x is a leaf) or recurse to search the appropriate subtree of x, after performing the necessary DISK-READ on that child.

Figure 18.1 illustrates the operation of B-TREE-SEARCH. The procedure examines the lightly shaded nodes during a search for the key R.

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. The B-TREE-SEARCH procedure therefore accesses $O(h) = O(\log_t n)$ disk pages, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $x.n < 2t$, the **while** loop of lines 2–3 takes $O(t)$ time within each node, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

To build a B-tree T, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in $O(1)$ time. We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.

```
B-TREE-CREATE$(T)$
1  $x = $ ALLOCATE-NODE()
2  $x.leaf = $ TRUE
3  $x.n = 0$
4  DISK-WRITE$(x)$
5  $T.root = x$
```

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

Inserting a key into a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree. As with binary search trees, we search for the leaf position at which to insert the new key. With a B-tree, however, we cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, we insert the new key into an existing leaf node. Since we cannot insert a key into a leaf node that is full, we introduce an operation that ***splits*** a full node y (having $2t - 1$ keys) around its ***median key*** $y.key_t$ into two nodes having only $t - 1$ keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, we must split it before we can insert the new key, and thus we could end up splitting full nodes all the way up the tree.

As with a binary search tree, we can insert a key into a B-tree in a single pass down the tree from the root to a leaf. To do so, we do not wait to find out whether we will actually need to split a full node in order to do the insertion. Instead, as we travel down the tree searching for the position where the new key belongs, we split each full node we come to along the way (including the leaf itself). Thus whenever we want to split a full node y, we are assured that its parent is not full.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes as input a *nonfull* internal node x (assumed to be in main memory) and an index i such that $x.c_i$ (also assumed to be in main memory) is a *full* child of x. The procedure then splits this child in two and adjusts x so that it has an additional child. To split a full root, we will first make the root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD. The tree thus grows in height by one; splitting is the only means by which the tree grows.

Figure 18.5 illustrates this process. We split the full node $y = x.c_i$ about its median key S, which moves up into y's parent node x. Those keys in y that are greater than the median key move into a new node z, which becomes a new child of x.

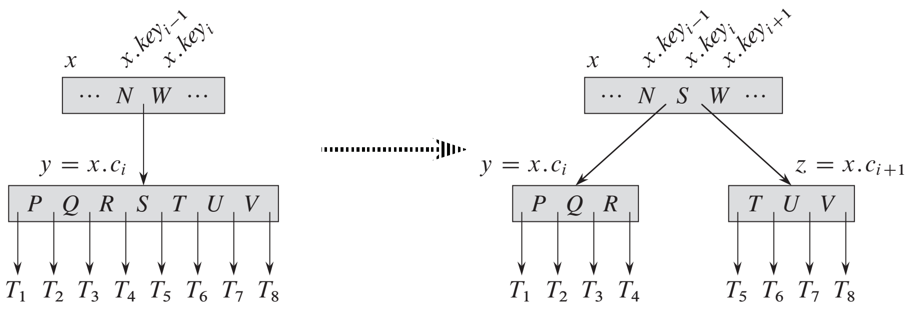


Figure 18.5 Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent.

```
B-TREE-SPLIT-CHILD$(x, i)$
 1  $z$ = ALLOCATE-NODE()
 2  $y = x.c_i$
 3  $z.leaf = y.leaf$
 4  $z.n = t - 1$
 5  for $j = 1$ to $t - 1$
 6      $z.key_j = y.key_{j+t}$
 7  if not $y.leaf$
 8      for $j = 1$ to $t$
 9          $z.c_j = y.c_{j+t}$
10  $y.n = t - 1$
11  for $j = x.n + 1$ downto $i + 1$
12      $x.c_{j+1} = x.c_j$
13  $x.c_{i+1} = z$
14  for $j = x.n$ downto $i$
15      $x.key_{j+1} = x.key_j$
16  $x.key_i = y.key_t$
17  $x.n = x.n + 1$
18  DISK-WRITE$(y)$
19  DISK-WRITE$(z)$
20  DISK-WRITE$(x)$
```

B-TREE-SPLIT-CHILD works by straightforward "cutting and pasting." Here, x is the node being split, and y is x's ith child (set in line 2). Node y originally has $2t$ children ($2t - 1$ keys) but is reduced to t children ($t - 1$ keys) by this operation. Node z takes the t largest children ($t - 1$ keys) from y, and z becomes a new child of x, positioned just after y in x's table of children. The median key of y moves up to become the key in x that separates y and z.

Lines 1–9 create node z and give it the largest $t - 1$ keys and corresponding t children of y. Line 10 adjusts the key count for y. Finally, lines 11–17 insert z as a child of x, move the median key from y up to x in order to separate y from z, and adjust x's key count. Lines 18–20 write out all modified disk pages. The CPU time used by B-Tree-Split-Child is $\Theta(t)$, due to the loops on lines 5–6 and 8–9. (The other loops run for $O(t)$ iterations.) The procedure performs $O(1)$ disk operations.

Inserting a key into a B-tree in a single pass down the tree

We insert a key k into a B-tree T of height h in a single pass down the tree, requiring $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$. The B-Tree-Insert procedure uses B-Tree-Split-Child to guarantee that the recursion never descends to a full node.

```
B-Tree-Insert(T, k)
 1  $r = T.root$
 2  **if** $r.n == 2t - 1$
 3      $s$ = Allocate-Node()
 4      $T.root = s$
 5      $s.leaf$ = FALSE
 6      $s.n = 0$
 7      $s.c_1 = r$
 8      B-Tree-Split-Child($s$, 1)
 9      B-Tree-Insert-Nonfull($s, k$)
10  **else** B-Tree-Insert-Nonfull($r, k$)
```

Lines 3–9 handle the case in which the root node r is full: the root splits and a new node s (having two children) becomes the root. Splitting the root is the only way to increase the height of a B-tree. Figure 18.6 illustrates this case. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom. The procedure finishes by calling B-Tree-Insert-Nonfull to insert key k into the tree rooted at the nonfull root node. B-Tree-Insert-Nonfull recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-Tree-Split-Child as necessary.

The auxiliary recursive procedure B-Tree-Insert-Nonfull inserts key k into node x, which is assumed to be nonfull when the procedure is called. The operation of B-Tree-Insert and the recursive operation of B-Tree-Insert-Nonfull guarantee that this assumption is true.

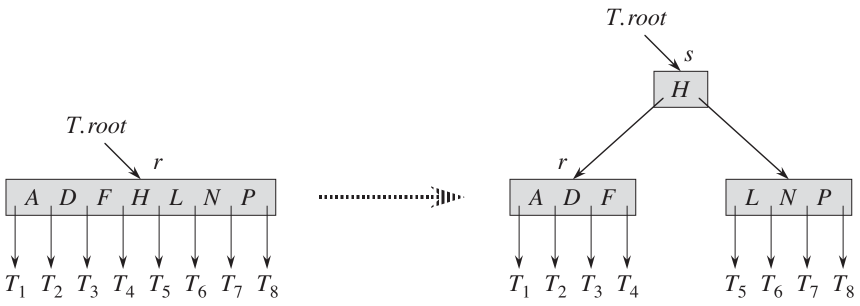


Figure 18.6 Splitting the root with $t = 4$. Root node r splits in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

```
B-TREE-INSERT-NONFULL$(x, k)$
 1  $i = x.n$
 2  **if** $x.leaf$
 3      **while** $i \geq 1$ and $k < x.key_i$
 4          $x.key_{i+1} = x.key_i$
 5          $i = i - 1$
 6      $x.key_{i+1} = k$
 7      $x.n = x.n + 1$
 8      DISK-WRITE$(x)$
 9  **else while** $i \geq 1$ and $k < x.key_i$
10          $i = i - 1$
11      $i = i + 1$
12      DISK-READ$(x.c_i)$
13      **if** $x.c_i.n == 2t - 1$
14          B-TREE-SPLIT-CHILD$(x, i)$
15          **if** $k > x.key_i$
16              $i = i + 1$
17      B-TREE-INSERT-NONFULL$(x.c_i, k)$
```

The B-TREE-INSERT-NONFULL procedure works as follows. Lines 3–8 handle the case in which x is a leaf node by inserting key k into x. If x is not a leaf node, then we must insert k into the appropriate leaf node in the subtree rooted at internal node x. In this case, lines 9–11 determine the child of x to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 uses B-TREE-SPLIT-CHILD to split that child into two nonfull children, and lines 15–16 determine which of the two children is now the correct one to descend to. (Note that there is no need for a DISK-READ$(x.c_i)$ after line 16 increments i, since the recursion will descend in this case to a child that was just created by B-TREE-SPLIT-CHILD.) The net effect of lines 13–16 is thus to guarantee that the procedure never recurses to a full node. Line 17 then recurses to insert k into the appropriate subtree. Figure 18.7 illustrates the various cases of inserting into a B-tree.

For a B-tree of height h, B-TREE-INSERT performs $O(h)$ disk accesses, since only $O(1)$ DISK-READ and DISK-WRITE operations occur between calls to B-TREE-INSERT-NONFULL. The total CPU time used is $O(th) = O(t \log_t n)$. Since B-TREE-INSERT-NONFULL is tail-recursive, we can alternatively implement it as a **while** loop, thereby demonstrating that the number of pages that need to be in main memory at any time is $O(1)$.

Exercises

18.2-1
Show the results of inserting the keys

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

18.2-2
Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)

18.2-3
Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

18.2-4 ★
Suppose that we insert the keys $\{1, 2, \ldots, n\}$ into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

18.2-5
Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

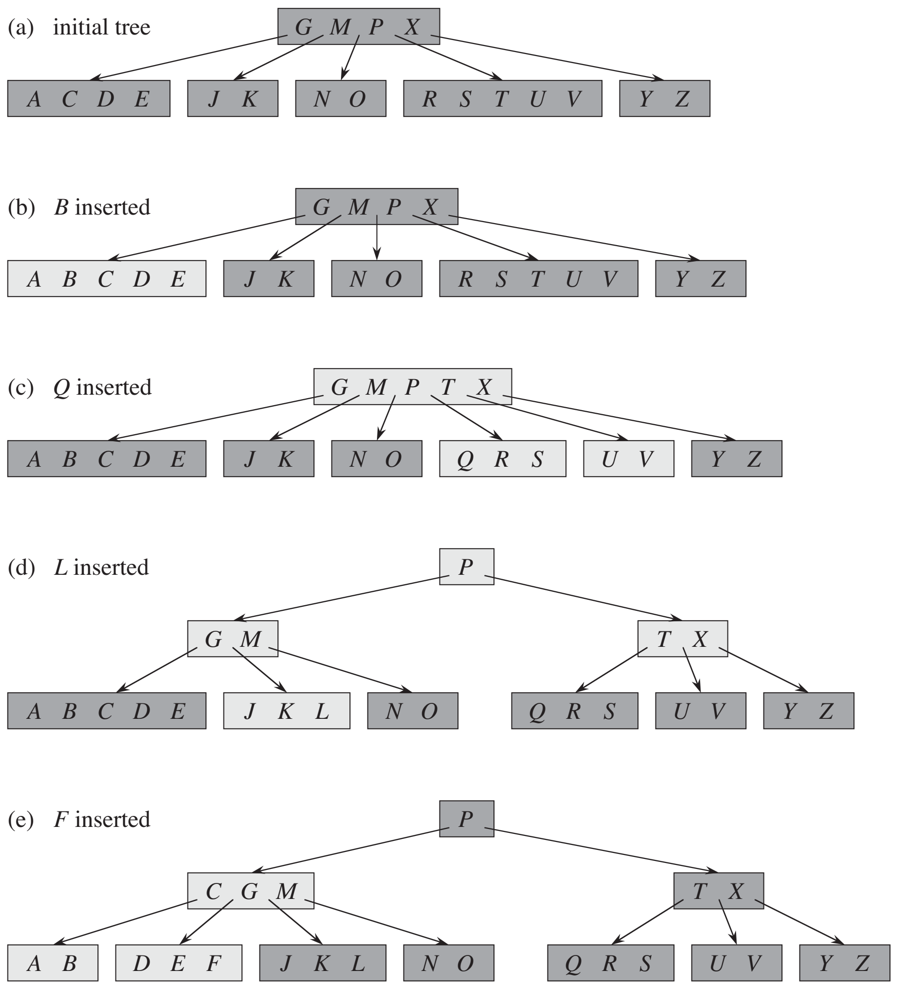


Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. **(a)** The initial tree for this example. **(b)** The result of inserting B into the initial tree; this is a simple insertion into a leaf node. **(c)** The result of inserting Q into the previous tree. The node $RSTUV$ splits into two nodes containing RS and UV, the key T moves up to the root, and Q is inserted in the leftmost of the two halves (the RS node). **(d)** The result of inserting L into the previous tree. The root splits right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. **(e)** The result of inserting F into the previous tree. The node $ABCDE$ splits before F is inserted into the rightmost of the two halves (the DE node).

18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n.

18.2-7

Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

18.3 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children. As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x. We design this procedure to guarantee that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t. Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b on pages 501–502), then we delete x, and x's only child $x.c_1$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

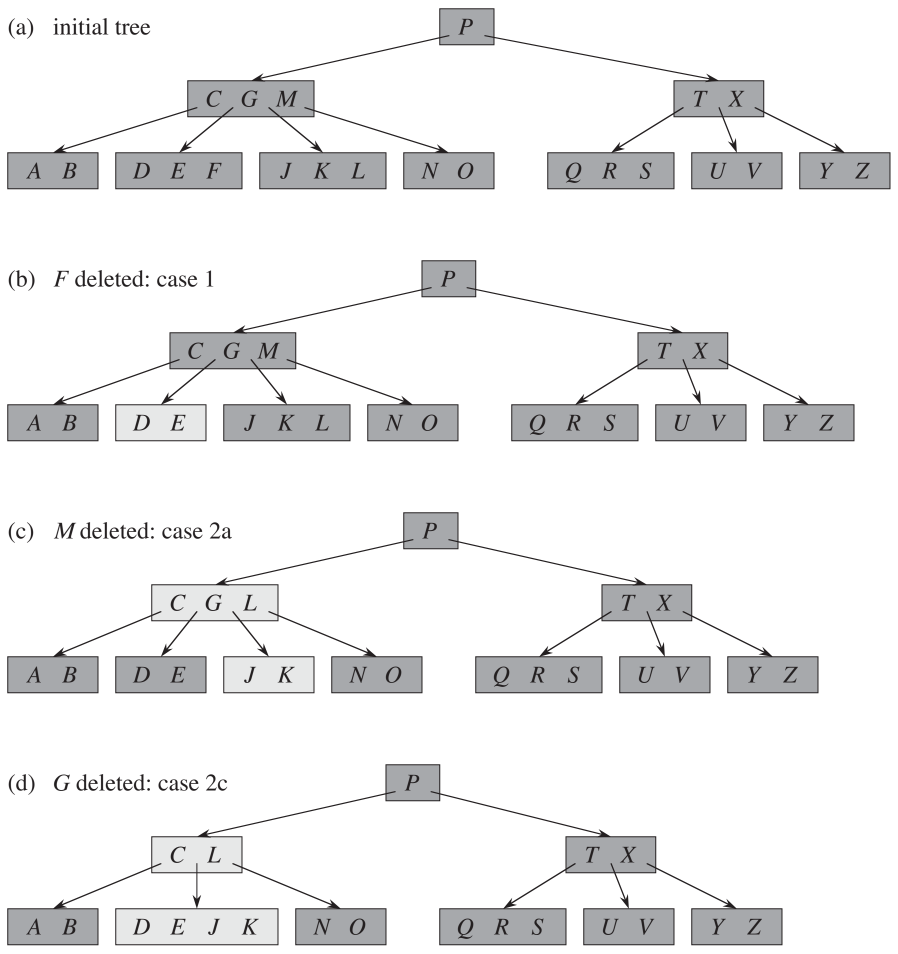


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F. This is case 1: simple deletion from a leaf. **(c)** Deletion of M. This is case 2a: the predecessor L of M moves up to take M's position. **(d)** Deletion of G. This is case 2c: we push G down to make node $DEGJK$ and then delete G from this leaf (case 1).

We sketch how deletion works instead of presenting the pseudocode. Figure 18.8 illustrates the various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following:

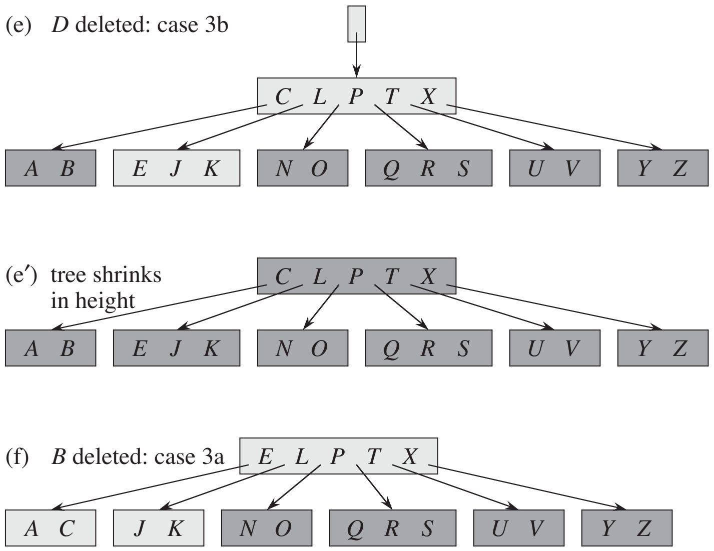


Figure 18.8, continued **(e)** Deletion of D. This is case 3b: the recursion cannot descend to node CL because it has only 2 keys, so we push P down and merge it with CL and TX to form $CLPTX$; then we delete D from a leaf (case 1). **(e′)** After (e), we delete the root and the tree shrinks in height by one. **(f)** Deletion of B. This is case 3a: C moves to fill B's position and E moves to fill C's position.

a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

c. Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t-1$ keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

a. If $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.

b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t-1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, we may expect that in practice, deletion operations are most often used to delete keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

Although this procedure seems complicated, it involves only $O(h)$ disk operations for a B-tree of height h, since only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure. The CPU time required is $O(th) = O(t \log_t n)$.

Exercises

18.3-1
Show the results of deleting C, P, and V, in order, from the tree of Figure 18.8(f).

18.3-2
Write pseudocode for B-TREE-DELETE.

Problems

18-1 Stacks on secondary storage
Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack we wish to support can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. We maintain in memory a stack pointer, which is the disk address of the top element on the stack. If the pointer has value p, the top element is the $(p \bmod m)$th word on page $\lfloor p/m \rfloor$ of the disk, where m is the number of words per page.

To implement the PUSH operation, we increment the stack pointer, read the appropriate page into memory from disk, copy the element to be pushed to the appropriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.

Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)

Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?

c. What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

18-2 Joining and splitting 2-3-4 trees

The ***join*** operation takes two dynamic sets S' and S'' and an element x such that for any $x' \in S'$ and $x'' \in S''$, we have $x'.key < x.key < x''.key$. It returns a set $S = S' \cup \{x\} \cup S''$. The ***split*** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and a set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. In this problem, we investigate how to implement these operations on 2-3-4 trees. We assume for convenience that elements consist only of keys and that all key values are distinct.

a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.

b. Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k, the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'', respectively.

c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k, the set S' of keys in T that are less than k, and the set S'' of keys in T that are greater than k. Show that p breaks S' into a set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$, where, for $i = 1, 2, \ldots, m$, we have $y < k'_i < z$ for any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i? Describe how p breaks S'' into sets of trees and keys.

d. Show how to implement the split operation on T. Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T''. The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T. (*Hint:* The costs for joining should telescope.)

Chapter notes

Knuth [211], Aho, Hopcroft, and Ullman [5], and Sedgewick [306] give further discussions of balanced-tree schemes and B-trees. Comer [74] provides a comprehensive survey of B-trees. Guibas and Sedgewick [155] discuss the relationships among various kinds of balanced-tree schemes, including red-black trees and 2-3-4 trees.

In 1970, J. E. Hopcroft invented 2-3 trees, a precursor to B-trees and 2-3-4 trees, in which every internal node has either two or three children. Bayer and McCreight [35] introduced B-trees in 1972; they did not explain their choice of name.

Bender, Demaine, and Farach-Colton [40] studied how to make B-trees perform well in the presence of memory-hierarchy effects. Their ***cache-oblivious*** algorithms work efficiently without explicitly knowing the data transfer sizes within the memory hierarchy.

Fibonacci Heaps and Binomial Heaps

19 Fibonacci Heaps

The Fibonacci heap data structure serves a dual purpose. First, it supports a set of operations that constitutes what is known as a "mergeable heap." Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

Mergeable heaps

A ***mergeable heap*** is any data structure that supports the following five operations, in which each element has a *key*:

MAKE-HEAP() creates and returns a new heap containing no elements.

INSERT(H, x) inserts element x, whose *key* has already been filled in, into heap H.

MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.

EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.

UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.[1]

DELETE(H, x) deletes element x from heap H.

[1]As mentioned in the introduction to Part V, our default mergeable heaps are mergeable min-heaps, and so the operations MINIMUM, EXTRACT-MIN, and DECREASE-KEY apply. Alternatively, we could define a ***mergeable max-heap*** with the operations MAXIMUM, EXTRACT-MAX, and INCREASE-KEY.

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Figure 19.1 Running times for operations on two implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by n.

As the table in Figure 19.1 shows, if we don't need the UNION operation, ordinary binary heaps, as used in heapsort (Chapter 6), work fairly well. Operations other than UNION run in worst-case time $O(\lg n)$ on a binary heap. If we need to support the UNION operation, however, binary heaps perform poorly. By concatenating the two arrays that hold the binary heaps to be merged and then running BUILD-MIN-HEAP (see Section 6.3), the UNION operation takes $\Theta(n)$ time in the worst case.

Fibonacci heaps, on the other hand, have better asymptotic time bounds than binary heaps for the INSERT, UNION, and DECREASE-KEY operations, and they have the same asymptotic running times for the remaining operations. Note, however, that the running times for Fibonacci heaps in Figure 19.1 are amortized time bounds, not worst-case per-operation time bounds. The UNION operation takes only constant amortized time in a Fibonacci heap, which is significantly better than the linear worst-case time required in a binary heap (assuming, of course, that an amortized time bound suffices).

Fibonacci heaps in theory and practice

From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed. This situation arises in many applications. For example, some algorithms for graph problems may call DECREASE-KEY once per edge. For dense graphs, which have many edges, the $\Theta(1)$ amortized time of each call of DECREASE-KEY adds up to a big improvement over the $\Theta(\lg n)$ worst-case time of binary heaps. Fast algorithms for problems such as computing minimum spanning trees (Chapter 23) and finding single-source shortest paths (Chapter 24) make essential use of Fibonacci heaps.

From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications, except for certain applications that manage large amounts of data. Thus, Fibonacci heaps are predominantly of theoretical interest. If a much simpler data structure with the same amortized time bounds as Fibonacci heaps were developed, it would be of practical use as well.

Both binary heaps and Fibonacci heaps are inefficient in how they support the operation SEARCH; it can take a while to find an element with a given key. For this reason, operations such as DECREASE-KEY and DELETE that refer to a given element require a pointer to that element as part of their input. As in our discussion of priority queues in Section 6.5, when we use a mergeable heap in an application, we often store a handle to the corresponding application object in each mergeable-heap element, as well as a handle to the corresponding mergeable-heap element in each application object. The exact nature of these handles depends on the application and its implementation.

Like several other data structures that we have seen, Fibonacci heaps are based on rooted trees. We represent each element by a node within a tree, and each node has a *key* attribute. For the remainder of this chapter, we shall use the term "node" instead of "element." We shall also ignore issues of allocating nodes prior to insertion and freeing nodes following deletion, assuming instead that the code calling the heap procedures deals with these details.

Section 19.1 defines Fibonacci heaps, discusses how we represent them, and presents the potential function used for their amortized analysis. Section 19.2 shows how to implement the mergeable-heap operations and achieve the amortized time bounds shown in Figure 19.1. The remaining two operations, DECREASE-KEY and DELETE, form the focus of Section 19.3. Finally, Section 19.4 finishes a key part of the analysis and also explains the curious name of the data structure.

19.1 Structure of Fibonacci heaps

A ***Fibonacci heap*** is a collection of rooted trees that are ***min-heap ordered***. That is, each tree obeys the ***min-heap property***: the key of a node is greater than or equal to the key of its parent. Figure 19.2(a) shows an example of a Fibonacci heap.

As Figure 19.2(b) shows, each node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the ***child list*** of x. Each child y in a child list has pointers $y.left$ and $y.right$ that point to y's left and right siblings, respectively. If node y is an only child, then $y.left = y.right = y$. Siblings may appear in a child list in any order.

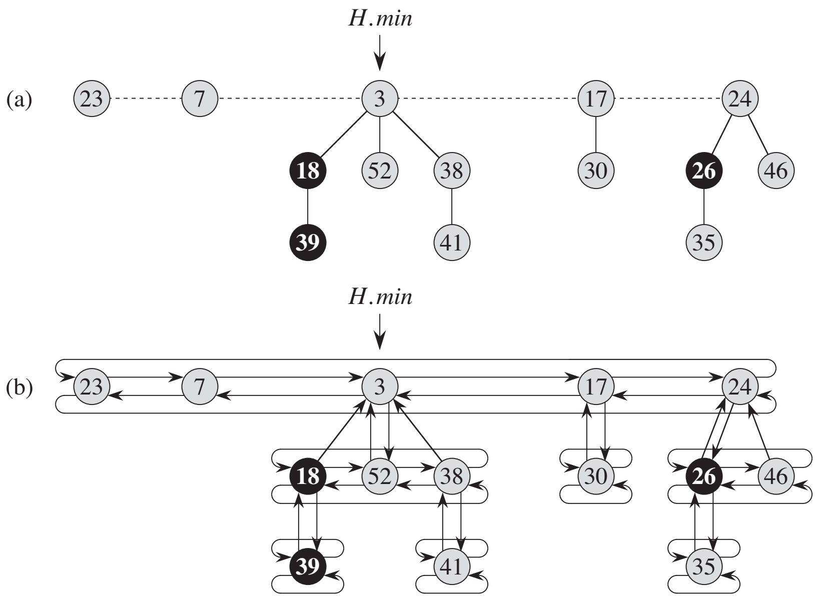


Figure 19.2 **(a)** A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. Black nodes are marked. The potential of this particular Fibonacci heap is $5 + 2 \cdot 3 = 11$. **(b)** A more complete representation showing pointers p (up arrows), *child* (down arrows), and *left* and *right* (sideways arrows). The remaining figures in this chapter omit these details, since all the information shown here can be determined from what appears in part (a).

Circular, doubly linked lists (see Section 10.2) have two advantages for use in Fibonacci heaps. First, we can insert a node into any location or remove a node from anywhere in a circular, doubly linked list in $O(1)$ time. Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in $O(1)$ time. In the descriptions of Fibonacci heap operations, we shall refer to these operations informally, letting you fill in the details of their implementations if you wish.

Each node has two other attributes. We store the number of children in the child list of node x in $x.degree$. The boolean-valued attribute $x.mark$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Until we look at the DECREASE-KEY operation in Section 19.3, we will just set all *mark* attributes to FALSE.

We access a given Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key; we call this node the ***minimum node*** of the Fibonacci heap. If more than one root has a key with the minimum value, then any such root may serve as the minimum node. When a Fibonacci heap H is empty, $H.min$ is NIL.

The roots of all the trees in a Fibonacci heap are linked together using their *left* and *right* pointers into a circular, doubly linked list called the ***root list*** of the Fibonacci heap. The pointer $H.min$ thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list.

We rely on one other attribute for a Fibonacci heap H: $H.n$, the number of nodes currently in H.

Potential function

As mentioned, we shall use the potential method of Section 17.3 to analyze the performance of Fibonacci heap operations. For a given Fibonacci heap H, we indicate by $t(H)$ the number of trees in the root list of H and by $m(H)$ the number of marked nodes in H. We then define the potential $\Phi(H)$ of Fibonacci heap H by

$$\Phi(H) = t(H) + 2\,m(H) . \tag{19.1}$$

(We will gain some intuition for this potential function in Section 19.3.) For example, the potential of the Fibonacci heap shown in Figure 19.2 is $5 + 2 \cdot 3 = 11$. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We shall assume that a unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter.

We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and by equation (19.1), the potential is nonnegative at all subsequent times. From equation (17.3), an upper bound on the total amortized cost provides an upper bound on the total actual cost for the sequence of operations.

Maximum degree

The amortized analyses we shall perform in the remaining sections of this chapter assume that we know an upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap. We won't prove it, but when only the mergeable-heap operations are supported, $D(n) \le \lfloor \lg n \rfloor$. (Problem 19-2(d) asks you to prove this property.) In Sections 19.3 and 19.4, we shall show that when we support DECREASE-KEY and DELETE as well, $D(n) = O(\lg n)$.

19.2 Mergeable-heap operations

The mergeable-heap operations on Fibonacci heaps delay work as long as possible. The various operations have performance trade-offs. For example, we insert a node by adding it to the root list, which takes just constant time. If we were to start with an empty Fibonacci heap and then insert k nodes, the Fibonacci heap would consist of just a root list of k nodes. The trade-off is that if we then perform an EXTRACT-MIN operation on Fibonacci heap H, after removing the node that $H.min$ points to, we would have to look through each of the remaining $k-1$ nodes in the root list to find the new minimum node. As long as we have to go through the entire root list during the EXTRACT-MIN operation, we also consolidate nodes into min-heap-ordered trees to reduce the size of the root list. We shall see that, no matter what the root list looks like before a EXTRACT-MIN operation, afterward each node in the root list has a degree that is unique within the root list, which leads to a root list of size at most $D(n) + 1$.

Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where $H.n = 0$ and $H.min =$ NIL; there are no trees in H. Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$. The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

Inserting a node

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that $x.key$ has already been filled in.

```
FIB-HEAP-INSERT(H, x)
 1  $x.degree = 0$
 2  $x.p =$ NIL
 3  $x.child =$ NIL
 4  $x.mark =$ FALSE
 5  if $H.min ==$ NIL
 6      create a root list for $H$ containing just $x$
 7      $H.min = x$
 8  else insert $x$ into $H$'s root list
 9      if $x.key < H.min.key$
10          $H.min = x$
11  $H.n = H.n + 1$
```

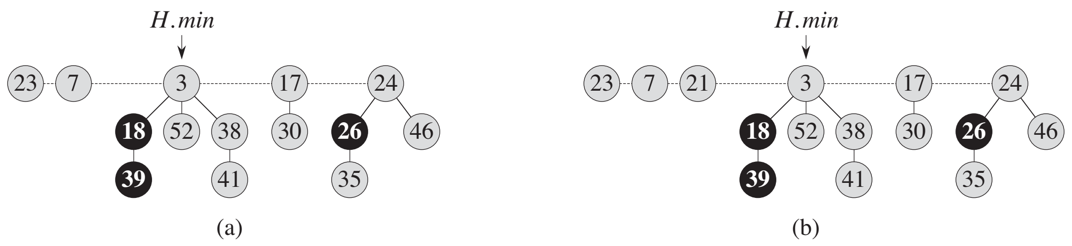


Figure 19.3 Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap H. **(b)** Fibonacci heap H after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

Lines 1–4 initialize some of the structural attributes of node x. Line 5 tests to see whether Fibonacci heap H is empty. If it is, then lines 6–7 make x be the only node in H's root list and set $H.min$ to point to x. Otherwise, lines 8–10 insert x into H's root list and update $H.min$ if necessary. Finally, line 11 increments $H.n$ to reflect the addition of the new node. Figure 19.3 shows a node with key 21 inserted into the Fibonacci heap of Figure 19.2.

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1 .$$

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer $H.min$, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H_1 and H_2, destroying H_1 and H_2 in the process. It simply concatenates the root lists of H_1 and H_2 and then determines the new minimum node. Afterward, the objects representing H_1 and H_2 will never be used again.

```
FIB-HEAP-UNION($H_1, H_2$)
1  $H$ = MAKE-FIB-HEAP()
2  $H.min = H_1.min$
3  concatenate the root list of $H_2$ with the root list of $H$
4  if ($H_1.min$ == NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
5      $H.min = H_2.min$
6  $H.n = H_1.n + H_2.n$
7  return $H$
```

Lines 1–3 concatenate the root lists of H_1 and H_2 into a new root list H. Lines 2, 4, and 5 set the minimum node of H, and line 6 sets $H.n$ to the total number of nodes. Line 7 returns the resulting Fibonacci heap H. As in the FIB-HEAP-INSERT procedure, all roots remain roots.

The change in potential is

$$\begin{aligned}
&\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\
&\quad = (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\
&\quad = 0 ,
\end{aligned}$$

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented in this section. It is also where the delayed work of consolidating trees in the root list finally occurs. The following pseudocode extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also calls the auxiliary procedure CONSOLIDATE, which we shall see shortly.

```
FIB-HEAP-EXTRACT-MIN(H)
 1  $z = H.min$
 2  if $z \neq$ NIL
 3      for each child $x$ of $z$
 4          add $x$ to the root list of $H$
 5          $x.p =$ NIL
 6      remove $z$ from the root list of $H$
 7      if $z == z.right$
 8          $H.min =$ NIL
 9      else $H.min = z.right$
10          CONSOLIDATE(H)
11      $H.n = H.n - 1$
12  return $z$
```

As Figure 19.4 illustrates, FIB-HEAP-EXTRACT-MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

We start in line 1 by saving a pointer z to the minimum node; the procedure returns this pointer at the end. If z is NIL, then Fibonacci heap H is already empty and we are done. Otherwise, we delete node z from H by making all of z's children roots of H in lines 3–5 (putting them into the root list) and removing z from the root list in line 6. If z is its own right sibling after line 6, then z was the only node on the root list and it had no children, so all that remains is to make the Fibonacci heap empty in line 8 before returning z. Otherwise, we set the pointer $H.min$ into the root list to point to a root other than z (in this case, z's right sibling), which is not necessarily going to be the new minimum node when FIB-HEAP-EXTRACT-MIN is done. Figure 19.4(b) shows the Fibonacci heap of Figure 19.4(a) after executing line 9.

The next step, in which we reduce the number of trees in the Fibonacci heap, is ***consolidating*** the root list of H, which the call CONSOLIDATE(H) accomplishes. Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct *degree* value:

1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
2. ***Link*** y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure. This procedure increments the attribute $x.degree$ and clears the mark on y.

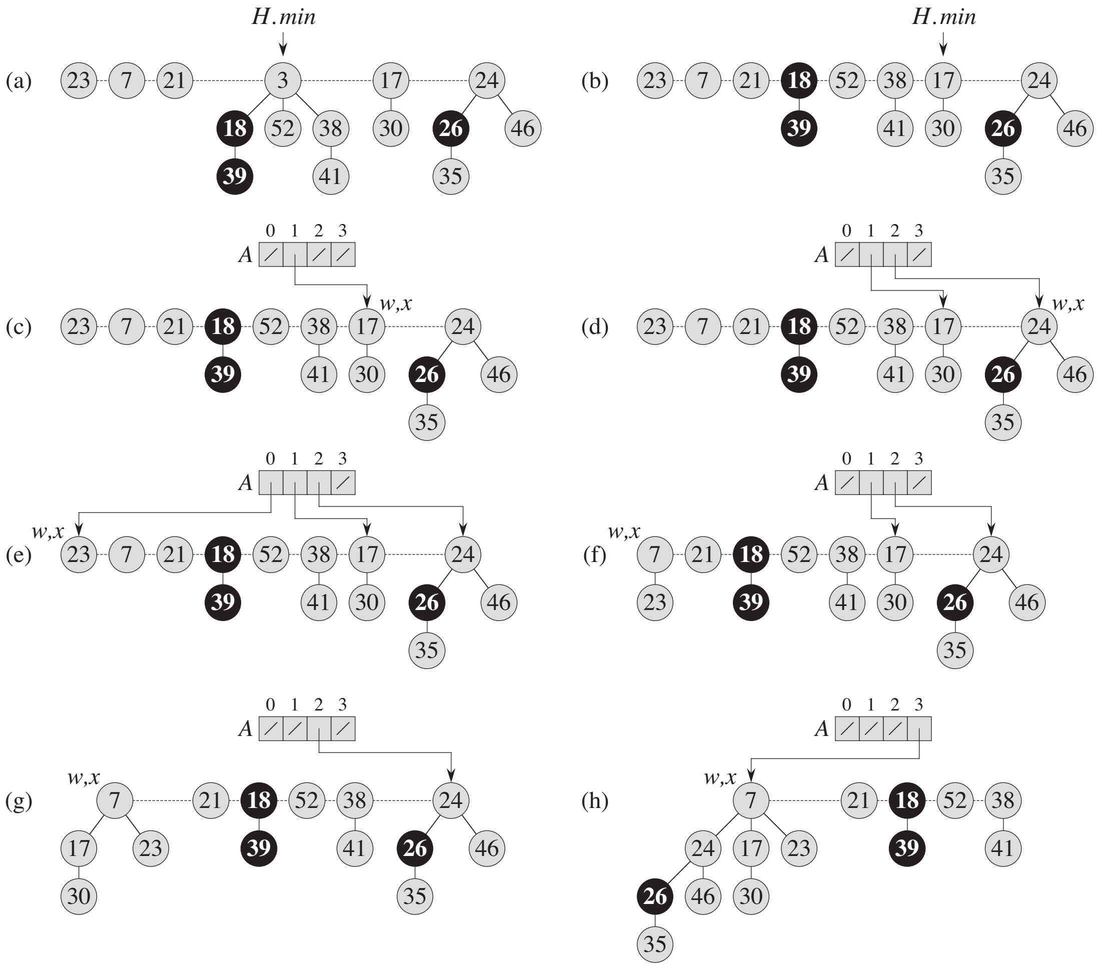


Figure 19.4 The action of FIB-HEAP-EXTRACT-MIN. **(a)** A Fibonacci heap H. **(b)** The situation after removing the minimum node z from the root list and adding its children to the root list. **(c)–(e)** The array A and the trees after each of the first three iterations of the **for** loop of lines 4–14 of the procedure CONSOLIDATE. The procedure processes the root list by starting at the node pointed to by $H.min$ and following *right* pointers. Each part shows the values of w and x at the end of an iteration. **(f)–(h)** The next iteration of the **for** loop, with the values of w and x shown at the end of each iteration of the **while** loop of lines 7–13. Part (f) shows the situation after the first time through the **while** loop. The node with key 23 has been linked to the node with key 7, which x now points to. In part (g), the node with key 17 has been linked to the node with key 7, which x still points to. In part (h), the node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by $A[3]$, at the end of the **for** loop iteration, $A[3]$ is set to point to the root of the resulting tree.

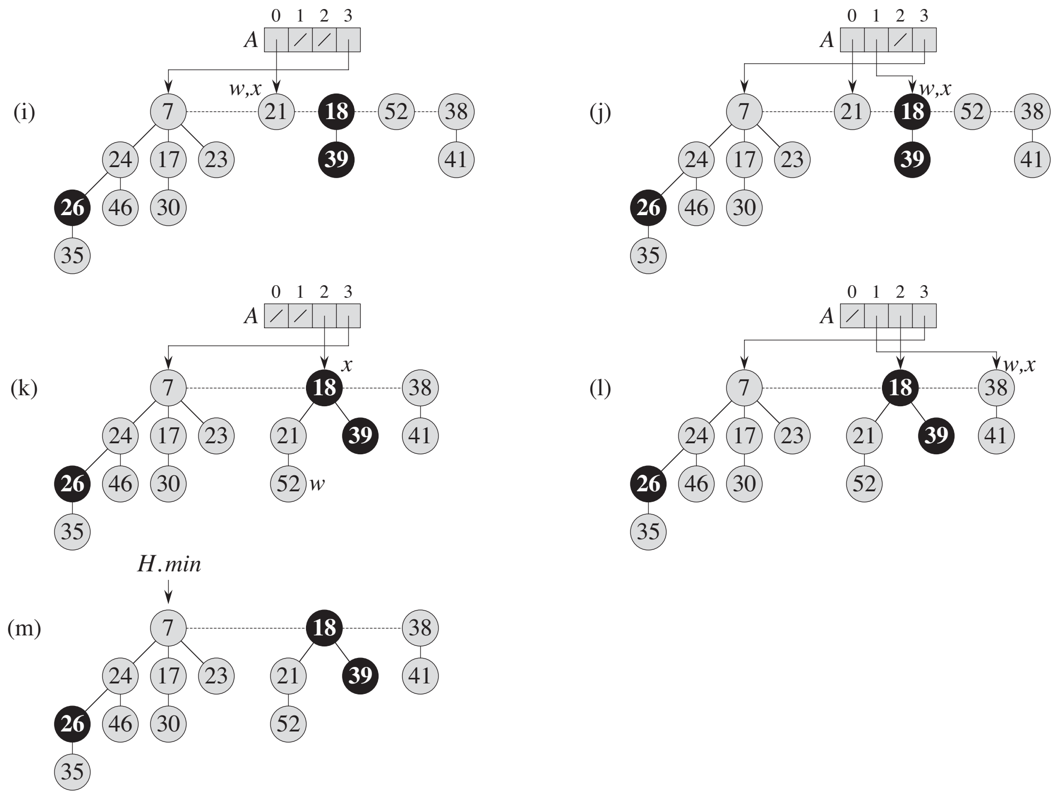


Figure 19.4, continued **(i)–(l)** The situation after each of the next four iterations of the **for** loop. **(m)** Fibonacci heap H after reconstructing the root list from the array A and determining the new $H.min$ pointer.

The procedure CONSOLIDATE uses an auxiliary array $A[0 \mathinner{..} D(H.n)]$ to keep track of roots according to their degrees. If $A[i] = y$, then y is currently a root with $y.degree = i$. Of course, in order to allocate the array we have to know how to calculate the upper bound $D(H.n)$ on the maximum degree, but we will see how to do so in Section 19.4.

```
CONSOLIDATE(H)
 1  let $A[0..D(H.n)]$ be a new array
 2  for $i = 0$ to $D(H.n)$
 3      $A[i] = $ NIL
 4  for each node $w$ in the root list of $H$
 5      $x = w$
 6      $d = x.degree$
 7      while $A[d] \neq$ NIL
 8          $y = A[d]$          // another node with the same degree as $x$
 9          if $x.key > y.key$
10              exchange $x$ with $y$
11          FIB-HEAP-LINK$(H, y, x)$
12          $A[d] = $ NIL
13          $d = d + 1$
14      $A[d] = x$
15  $H.min = $ NIL
16  for $i = 0$ to $D(H.n)$
17      if $A[i] \neq$ NIL
18          if $H.min ==$ NIL
19              create a root list for $H$ containing just $A[i]$
20              $H.min = A[i]$
21          else insert $A[i]$ into $H$'s root list
22              if $A[i].key < H.min.key$
23                  $H.min = A[i]$
```

```
FIB-HEAP-LINK$(H, y, x)$
1  remove $y$ from the root list of $H$
2  make $y$ a child of $x$, incrementing $x.degree$
3  $y.mark = $ FALSE
```

In detail, the CONSOLIDATE procedure works as follows. Lines 1–3 allocate and initialize the array A by making each entry NIL. The **for** loop of lines 4–14 processes each root w in the root list. As we link roots together, w may be linked to some other node and no longer be a root. Nevertheless, w is always in a tree rooted at some node x, which may or may not be w itself. Because we want at most one root with each degree, we look in the array A to see whether it contains a root y with the same degree as x. If it does, then we link the roots x and y but guaranteeing that x remains a root after linking. That is, we link y to x after first exchanging the pointers to the two roots if y's key is smaller than x's key. After we link y to x, the degree of x has increased by 1, and so we continue this process, linking x and another root whose degree equals x's new degree, until no other root that we have processed has the same degree as x. We then set the appropriate entry of A to point to x, so that as we process roots later on, we have recorded that x is the unique root of its degree that we have already processed. When this **for** loop terminates, at most one root of each degree will remain, and the array A will point to each remaining root.

The **while** loop of lines 7–13 repeatedly links the root x of the tree containing node w to another tree whose root has the same degree as x, until no other root has the same degree. This **while** loop maintains the following invariant:

At the start of each iteration of the **while** loop, $d = x.degree$.

We use this loop invariant as follows:

Initialization: Line 6 ensures that the loop invariant holds the first time we enter the loop.

Maintenance: In each iteration of the **while** loop, $A[d]$ points to some root y. Because $d = x.degree = y.degree$, we want to link x and y. Whichever of x and y has the smaller key becomes the parent of the other as a result of the link operation, and so lines 9–10 exchange the pointers to x and y if necessary. Next, we link y to x by the call FIB-HEAP-LINK(H, y, x) in line 11. This call increments $x.degree$ but leaves $y.degree$ as d. Node y is no longer a root, and so line 12 removes the pointer to it in array A. Because the call of FIB-HEAP-LINK increments the value of $x.degree$, line 13 restores the invariant that $d = x.degree$.

Termination: We repeat the **while** loop until $A[d] = \text{NIL}$, in which case there is no other root with the same degree as x.

After the **while** loop terminates, we set $A[d]$ to x in line 14 and perform the next iteration of the **for** loop.

Figures 19.4(c)–(e) show the array A and the resulting trees after the first three iterations of the **for** loop of lines 4–14. In the next iteration of the **for** loop, three links occur; their results are shown in Figures 19.4(f)–(h). Figures 19.4(i)–(l) show the result of the next four iterations of the **for** loop.

All that remains is to clean up. Once the **for** loop of lines 4–14 completes, line 15 empties the root list, and lines 16–23 reconstruct it from the array A. The resulting Fibonacci heap appears in Figure 19.4(m). After consolidating the root list, FIB-HEAP-EXTRACT-MIN finishes up by decrementing $H.n$ in line 11 and returning a pointer to the deleted node z in line 12.

We are now ready to show that the amortized cost of extracting the minimum node of an n-node Fibonacci heap is $O(D(n))$. Let H denote the Fibonacci heap just prior to the FIB-HEAP-EXTRACT-MIN operation.

We start by accounting for the actual cost of extracting the minimum node. An $O(D(n))$ contribution comes from FIB-HEAP-EXTRACT-MIN processing at most $D(n)$ children of the minimum node and from the work in lines 2–3 and 16–23 of CONSOLIDATE. It remains to analyze the contribution from the **for** loop of lines 4–14 in CONSOLIDATE, for which we use an aggregate analysis. The size of the root list upon calling CONSOLIDATE is at most $D(n) + t(H) - 1$, since it consists of the original $t(H)$ root-list nodes, minus the extracted root node, plus the children of the extracted node, which number at most $D(n)$. Within a given iteration of the **for** loop of lines 4–14, the number of iterations of the **while** loop of lines 7–13 depends on the root list. But we know that every time through the **while** loop, one of the roots is linked to another, and thus the total number of iterations of the **while** loop over all iterations of the **for** loop is at most the number of roots in the root list. Hence, the total amount of work performed in the **for** loop is at most proportional to $D(n) + t(H)$. Thus, the total actual work in extracting the minimum node is $O(D(n) + t(H))$.

The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential afterward is at most $(D(n) + 1) + 2m(H)$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation. The amortized cost is thus at most

$$\begin{aligned}
&O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\
&\qquad = O(D(n)) + O(t(H)) - t(H) \\
&\qquad = O(D(n)) ,
\end{aligned}$$

since we can scale up the units of potential to dominate the constant hidden in $O(t(H))$. Intuitively, the cost of performing each link is paid for by the reduction in potential due to the link's reducing the number of roots by one. We shall see in Section 19.4 that $D(n) = O(\lg n)$, so that the amortized cost of extracting the minimum node is $O(\lg n)$.

Exercises

19.2-1
Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on the Fibonacci heap shown in Figure 19.4(m).

19.3 Decreasing a key and deleting a node

In this section, we show how to decrease the key of a node in a Fibonacci heap in $O(1)$ amortized time and how to delete any node from an n-node Fibonacci heap in $O(D(n))$ amortized time. In Section 19.4, we will show that the maximum degree $D(n)$ is $O(\lg n)$, which will imply that FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELETE run in $O(\lg n)$ amortized time.

Decreasing a key

In the following pseudocode for the operation FIB-HEAP-DECREASE-KEY, we assume as before that removing a node from a linked list does not change any of the structural attributes in the removed node.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if $k > x.key$
2      error "new key is greater than current key"
3  $x.key = k$
4  $y = x.p$
5  if $y \neq$ NIL and $x.key < y.key$
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if $x.key < H.min.key$
9      $H.min = x$
```

```
CUT(H, x, y)
1  remove $x$ from the child list of $y$, decrementing $y.degree$
2  add $x$ to the root list of $H$
3  $x.p$ = NIL
4  $x.mark$ = FALSE
```

```
CASCADING-CUT(H, y)
1  $z = y.p$
2  if $z \neq$ NIL
3      if $y.mark$ == FALSE
4          $y.mark$ = TRUE
5      else CUT(H, y, z)
6          CASCADING-CUT(H, z)
```

The FIB-HEAP-DECREASE-KEY procedure works as follows. Lines 1–3 ensure that the new key is no greater than the current key of x and then assign the new key to x. If x is a root or if $x.key \geq y.key$, where y is x's parent, then no structural changes need occur, since min-heap order has not been violated. Lines 4–5 test for this condition.

If min-heap order has been violated, many changes may occur. We start by ***cutting*** x in line 6. The CUT procedure "cuts" the link between x and its parent y, making x a root.

We use the *mark* attributes to obtain the desired time bounds. They record a little piece of the history of each node. Suppose that the following events have happened to node x:

1. at some time, x was a root,
2. then x was linked to (made the child of) another node,
3. then two children of x were removed by cuts.

As soon as the second child has been lost, we cut x from its parent, making it a new root. The attribute $x.mark$ is TRUE if steps 1 and 2 have occurred and one child of x has been cut. The CUT procedure, therefore, clears $x.mark$ in line 4, since it performs step 1. (We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$: node y is being linked to another node, and so step 2 is being performed. The next time a child of y is cut, $y.mark$ will be set to TRUE.)

We are not yet done, because x might be the second child cut from its parent y since the time that y was linked to another node. Therefore, line 7 of FIB-HEAP-DECREASE-KEY attempts to perform a ***cascading-cut*** operation on y. If y is a root, then the test in line 2 of CASCADING-CUT causes the procedure to just return. If y is unmarked, the procedure marks it in line 4, since its first child has just been cut, and returns. If y is marked, however, it has just lost its second child; y is cut in line 5, and CASCADING-CUT calls itself recursively in line 6 on y's parent z. The CASCADING-CUT procedure recurses its way up the tree until it finds either a root or an unmarked node.

Once all the cascading cuts have occurred, lines 8–9 of FIB-HEAP-DECREASE-KEY finish up by updating $H.min$ if necessary. The only node whose key changed was the node x whose key decreased. Thus, the new minimum node is either the original minimum node or node x.

Figure 19.5 shows the execution of two calls of FIB-HEAP-DECREASE-KEY, starting with the Fibonacci heap shown in Figure 19.5(a). The first call, shown in Figure 19.5(b), involves no cascading cuts. The second call, shown in Figures 19.5(c)–(e), invokes two cascading cuts.

We shall now show that the amortized cost of FIB-HEAP-DECREASE-KEY is only $O(1)$. We start by determining its actual cost. The FIB-HEAP-DECREASE-KEY procedure takes $O(1)$ time, plus the time to perform the cascading cuts. Suppose that a given invocation of FIB-HEAP-DECREASE-KEY results in c calls of CASCADING-CUT (the call made from line 7 of FIB-HEAP-DECREASE-KEY followed by $c - 1$ recursive calls of CASCADING-CUT). Each call of CASCADING-CUT takes $O(1)$ time exclusive of recursive calls. Thus, the actual cost of FIB-HEAP-DECREASE-KEY, including all recursive calls, is $O(c)$.

We next compute the change in potential. Let H denote the Fibonacci heap just prior to the FIB-HEAP-DECREASE-KEY operation. The call to CUT in line 6 of

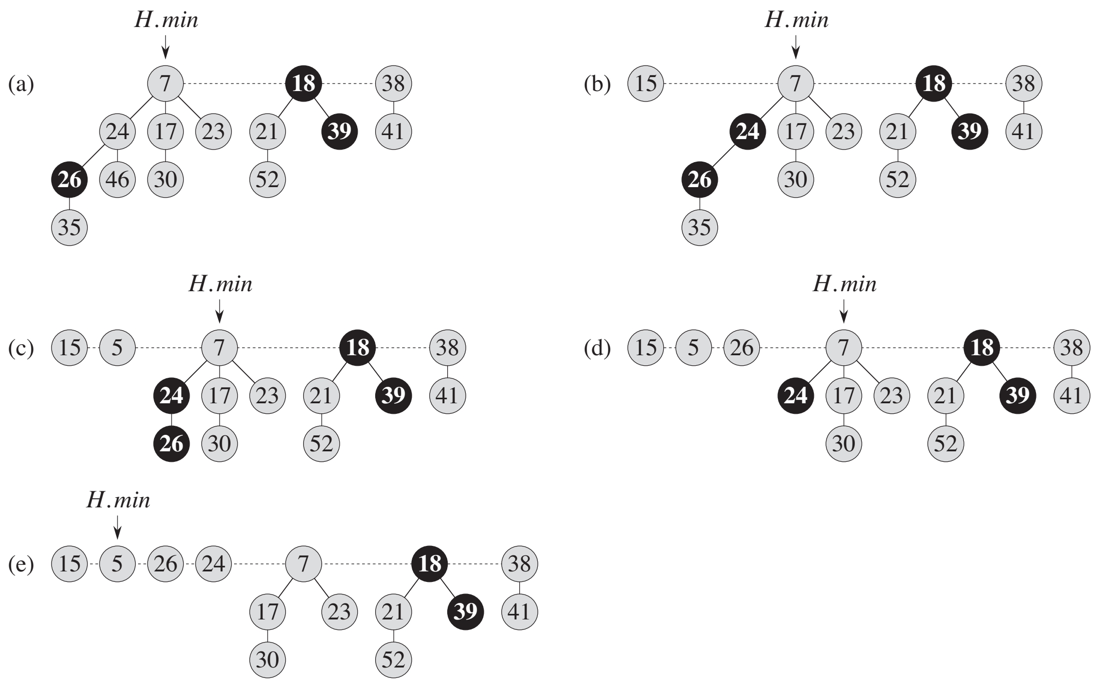


Figure 19.5 Two calls of FIB-HEAP-DECREASE-KEY. **(a)** The initial Fibonacci heap. **(b)** The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. **(c)–(e)** The node with key 35 has its key decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) Part (e) shows the result of the FIB-HEAP-DECREASE-KEY operation, with $H.min$ pointing to the new minimum node.

FIB-HEAP-DECREASE-KEY creates a new tree rooted at node x and clears x's mark bit (which may have already been FALSE). Each call of CASCADING-CUT, except for the last one, cuts a marked node and clears the mark bit. Afterward, the Fibonacci heap contains $t(H)+c$ trees (the original $t(H)$ trees, $c-1$ trees produced by cascading cuts, and the tree rooted at x) and at most $m(H)-c+2$ marked nodes ($c-1$ were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node). The change in potential is therefore at most

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H)) = 4-c\ .$$

Thus, the amortized cost of FIB-HEAP-DECREASE-KEY is at most

$$O(c) + 4 - c = O(1) ,$$

since we can scale up the units of potential to dominate the constant hidden in $O(c)$.

You can now see why we defined the potential function to include a term that is twice the number of marked nodes. When a marked node y is cut by a cascading cut, its mark bit is cleared, which reduces the potential by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y becoming a root.

Deleting a node

The following pseudocode deletes a node from an n-node Fibonacci heap in $O(D(n))$ amortized time. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE makes x become the minimum node in the Fibonacci heap by giving it a uniquely small key of $-\infty$. The FIB-HEAP-EXTRACT-MIN procedure then removes node x from the Fibonacci heap. The amortized time of FIB-HEAP-DELETE is the sum of the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY and the $O(D(n))$ amortized time of FIB-HEAP-EXTRACT-MIN. Since we shall see in Section 19.4 that $D(n) = O(\lg n)$, the amortized time of FIB-HEAP-DELETE is $O(\lg n)$.

Exercises

19.3-1
Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a marked root. Argue that it doesn't matter to the analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.

19.3-2
Justify the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY as an average cost per operation by using aggregate analysis.

19.4 Bounding the maximum degree

To prove that the amortized time of FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELETE is $O(\lg n)$, we must show that the upper bound $D(n)$ on the degree of any node of an n-node Fibonacci heap is $O(\lg n)$. In particular, we shall show that $D(n) \le \lfloor \log_\phi n \rfloor$, where ϕ is the golden ratio, defined in equation (3.24) as

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots .$$

The key to the analysis is as follows. For each node x within a Fibonacci heap, define size(x) to be the number of nodes, including x itself, in the subtree rooted at x. (Note that x need not be in the root list—it can be any node at all.) We shall show that size(x) is exponential in $x.degree$. Bear in mind that $x.degree$ is always maintained as an accurate count of the degree of x.

Lemma 19.1
Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $y_1.degree \ge 0$ and $y_i.degree \ge i - 2$ for $i = 2, 3, \ldots, k$.

Proof Obviously, $y_1.degree \ge 0$.

For $i \ge 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, and so we must have had $x.degree \ge i - 1$. Because node y_i is linked to x (by CONSOLIDATE) only if $x.degree = y_i.degree$, we must have also had $y_i.degree \ge i - 1$ at that time. Since then, node y_i has lost at most one child, since it would have been cut from x (by CASCADING-CUT) if it had lost two children. We conclude that $y_i.degree \ge i - 2$. ■

We finally come to the part of the analysis that explains the name "Fibonacci heaps." Recall from Section 3.2 that for $k = 0, 1, 2, \ldots$, the kth Fibonacci number is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k = 0 , \\ 1 & \text{if } k = 1 , \\ F_{k-1} + F_{k-2} & \text{if } k \ge 2 . \end{cases}$$

The following lemma gives another way to express F_k.

Lemma 19.2
For all integers $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \ .$$

Proof The proof is by induction on k. When $k = 0$,

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= F_2 \ . \end{aligned}$$

We now assume the inductive hypothesis that $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, and we have

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i \ . \end{aligned}$$ ■

Lemma 19.3
For all integers $k \geq 0$, the $(k + 2)$nd Fibonacci number satisfies $F_{k+2} \geq \phi^k$.

Proof The proof is by induction on k. The base cases are for $k = 0$ and $k = 1$. When $k = 0$ we have $F_2 = 1 = \phi^0$, and when $k = 1$ we have $F_3 = 2 > 1.619 > \phi^1$. The inductive step is for $k \geq 2$, and we assume that $F_{i+2} > \phi^i$ for $i = 0, 1, \ldots, k-1$. Recall that ϕ is the positive root of equation (3.23), $x^2 = x+1$. Thus, we have

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &\geq \phi^{k-1} + \phi^{k-2} && \text{(by the inductive hypothesis)} \\ &= \phi^{k-2}(\phi + 1) \\ &= \phi^{k-2} \cdot \phi^2 && \text{(by equation (3.23))} \\ &= \phi^k \ . \end{aligned}$$ ■

The following lemma and its corollary complete the analysis.

Lemma 19.4
Let x be any node in a Fibonacci heap, and let $k = x.degree$. Then size$(x) \geq F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5})/2$.

Proof Let s_k denote the minimum possible size of any node of degree k in any Fibonacci heap. Trivially, $s_0 = 1$ and $s_1 = 2$. The number s_k is at most size(x) and, because adding children to a node cannot decrease the node's size, the value of s_k increases monotonically with k. Consider some node z, in any Fibonacci heap, such that $z.degree = k$ and size$(z) = s_k$. Because $s_k \leq$ size(x), we compute a lower bound on size(x) by computing a lower bound on s_k. As in Lemma 19.1, let $y_1, y_2, \ldots, y_k$ denote the children of z in the order in which they were linked to z. To bound s_k, we count one for z itself and one for the first child y_1 (for which size$(y_1) \geq 1$), giving

$$\begin{aligned} \text{size}(x) &\geq s_k \\ &\geq 2 + \sum_{i=2}^{k} s_{y_i.degree} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \;, \end{aligned}$$

where the last line follows from Lemma 19.1 (so that $y_i.degree \geq i - 2$) and the monotonicity of s_k (so that $s_{y_i.degree} \geq s_{i-2}$).

We now show by induction on k that $s_k \geq F_{k+2}$ for all nonnegative integers k. The bases, for $k = 0$ and $k = 1$, are trivial. For the inductive step, we assume that $k \geq 2$ and that $s_i \geq F_{i+2}$ for $i = 0, 1, \ldots, k - 1$. We have

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} && \text{(by Lemma 19.2)} \\ &\geq \phi^k && \text{(by Lemma 19.3)} \;. \end{aligned}$$

Thus, we have shown that size$(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

Corollary 19.5
The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\lg n)$.

Proof Let x be any node in an n-node Fibonacci heap, and let $k = x.degree$. By Lemma 19.4, we have $n \geq \text{size}(x) \geq \phi^k$. Taking base-$\phi$ logarithms gives us $k \leq \log_\phi n$. (In fact, because k is an integer, $k \leq \lfloor \log_\phi n \rfloor$.) The maximum degree $D(n)$ of any node is thus $O(\lg n)$. ■

Exercises

19.4-1
Professor Pinocchio claims that the height of an n-node Fibonacci heap is $O(\lg n)$. Show that the professor is mistaken by exhibiting, for any positive integer n, a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

19.4-2
Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. (The rule in Section 19.3 uses $k = 2$.) For what values of k is $D(n) = O(\lg n)$?

Problems

19-1 Alternative implementation of deletion
Professor Pisano has proposed the following variant of the FIB-HEAP-DELETE procedure, claiming that it runs faster when the node being deleted is not the node pointed to by $H.min$.

```
PISANO-DELETE(H, x)
1  if x == H.min
2      FIB-HEAP-EXTRACT-MIN(H)
3  else y = x.p
4      if y ≠ NIL
5          CUT(H, x, y)
6          CASCADING-CUT(H, y)
7      add x's child list to the root list of H
8      remove x from the root list of H
```

a. The professor's claim that this procedure runs faster is based partly on the assumption that line 7 can be performed in $O(1)$ actual time. What is wrong with this assumption?

b. Give a good upper bound on the actual time of PISANO-DELETE when x is not $H.min$. Your bound should be in terms of $x.degree$ and the number c of calls to the CASCADING-CUT procedure.

c. Suppose that we call PISANO-DELETE(H, x), and let H' be the Fibonacci heap that results. Assuming that node x is not a root, bound the potential of H' in terms of $x.degree$, c, $t(H)$, and $m(H)$.

d. Conclude that the amortized time for PISANO-DELETE is asymptotically no better than for FIB-HEAP-DELETE, even when $x \neq H.min$.

19-2 Binomial trees and binomial heaps

The ***binomial tree*** B_k is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.6(a), the binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together so that the root of one is the leftmost child of the root of the other. Figure 19.6(b) shows the binomial trees B_0 through B_4.

a. Show that for the binomial tree B_k,

1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$, and
4. the root has degree k, which is greater than that of any other node; moreover, as Figure 19.6(c) shows, if we number the children of the root from left to right by $k-1, k-2, \ldots, 0$, then child i is the root of a subtree B_i.

A ***binomial heap*** H is a set of binomial trees that satisfies the following properties:

1. Each node has a *key* (like a Fibonacci heap).
2. Each binomial tree in H obeys the min-heap property.
3. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

b. Suppose that a binomial heap H has a total of n nodes. Discuss the relationship between the binomial trees that H contains and the binary representation of n. Conclude that H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

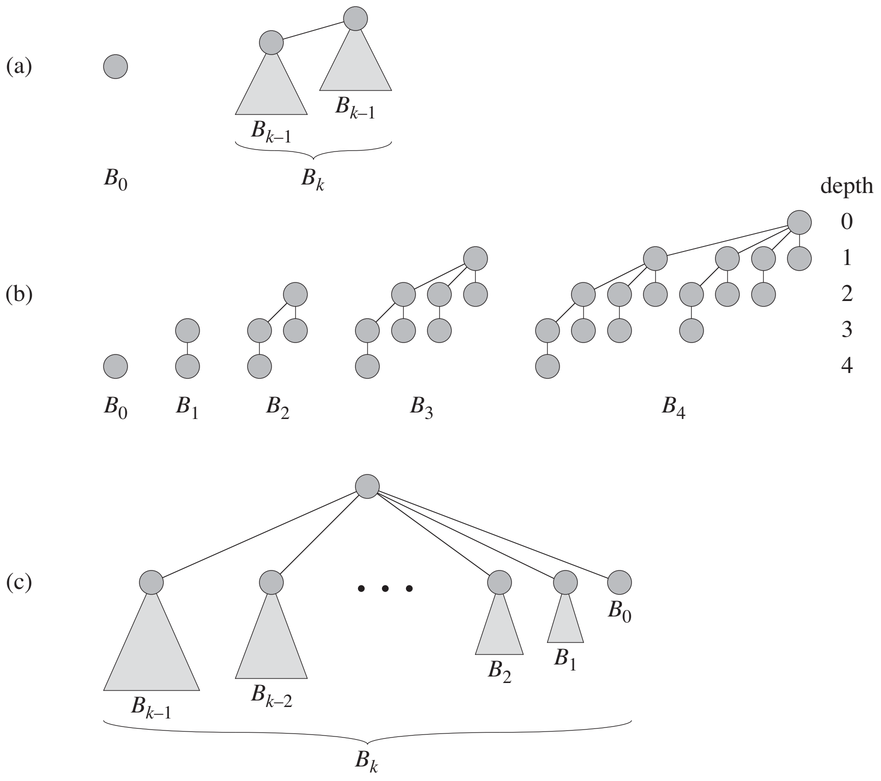


Figure 19.6 **(a)** The recursive definition of the binomial tree B_k. Triangles represent rooted subtrees. **(b)** The binomial trees B_0 through B_4. Node depths in B_4 are shown. **(c)** Another way of looking at the binomial tree B_k.

Suppose that we represent a binomial heap as follows. The left-child, right-sibling scheme of Section 10.4 represents each binomial tree within a binomial heap. Each node contains its key; pointers to its parent, to its leftmost child, and to the sibling immediately to its right (these pointers are NIL when appropriate); and its degree (as in Fibonacci heaps, how many children it has). The roots form a singly linked root list, ordered by the degrees of the roots (from low to high), and we access the binomial heap by a pointer to the first node on the root list.

c. Complete the description of how to represent a binomial heap (i.e., name the attributes, describe when attributes have the value NIL, and define how the root list is organized), and show how to implement the same seven operations on binomial heaps as this chapter implemented on Fibonacci heaps. Each operation should run in $O(\lg n)$ worst-case time, where n is the number of nodes in the binomial heap (or in the case of the UNION operation, in the two binomial heaps that are being united). The MAKE-HEAP operation should take constant time.

d. Suppose that we were to implement only the mergeable-heap operations on a Fibonacci heap (i.e., we do not implement the DECREASE-KEY or DELETE operations). How would the trees in a Fibonacci heap resemble those in a binomial heap? How would they differ? Show that the maximum degree in an n-node Fibonacci heap would be at most $\lfloor \lg n \rfloor$.

e. Professor McGee has devised a new data structure based on Fibonacci heaps. A McGee heap has the same structure as a Fibonacci heap and supports just the mergeable-heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union consolidate the root list as their last step. What are the worst-case running times of operations on McGee heaps?

19-3 More Fibonacci-heap operations

We wish to augment a Fibonacci heap H to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.

a. The operation FIB-HEAP-CHANGE-KEY(H, x, k) changes the key of node x to the value k. Give an efficient implementation of FIB-HEAP-CHANGE-KEY, and analyze the amortized running time of your implementation for the cases in which k is greater than, less than, or equal to $x.key$.

b. Give an efficient implementation of FIB-HEAP-PRUNE(H, r), which deletes $q = \min(r, H.n)$ nodes from H. You may choose any q nodes to delete. Analyze the amortized running time of your implementation. (*Hint:* You may need to modify the data structure and potential function.)

19-4 2-3-4 heaps

Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. In this problem, we shall implement ***2-3-4 heaps***, which support the mergeable-heap operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps, only leaves store keys, and each leaf x stores exactly one key in the attribute $x.key$. The keys in the leaves may appear in any order. Each internal node x contains a value $x.small$ that is equal to the smallest key stored in any leaf in the subtree rooted at x. The root r contains an attribute $r.height$ that gives the height of the tree. Finally, 2-3-4 heaps are designed to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. In parts (a)–(e), each operation should run in $O(\lg n)$ time on a 2-3-4 heap with n elements. The UNION operation in part (f) should run in $O(\lg n)$ time, where n is the number of elements in the two input heaps.

a. MINIMUM, which returns a pointer to the leaf with the smallest key.

b. DECREASE-KEY, which decreases the key of a given leaf x to a given value $k \le x.key$.

c. INSERT, which inserts leaf x with key k.

d. DELETE, which deletes a given leaf x.

e. EXTRACT-MIN, which extracts the leaf with the smallest key.

f. UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and destroying the input heaps.

Chapter notes

Fredman and Tarjan [114] introduced Fibonacci heaps. Their paper also describes the application of Fibonacci heaps to the problems of single-source shortest paths, all-pairs shortest paths, weighted bipartite matching, and the minimum-spanning-tree problem.

Subsequently, Driscoll, Gabow, Shrairman, and Tarjan [96] developed "relaxed heaps" as an alternative to Fibonacci heaps. They devised two varieties of relaxed heaps. One gives the same amortized time bounds as Fibonacci heaps. The other allows DECREASE-KEY to run in $O(1)$ worst-case (not amortized) time and EXTRACT-MIN and DELETE to run in $O(\lg n)$ worst-case time. Relaxed heaps also have some advantages over Fibonacci heaps in parallel algorithms.

See also the chapter notes for Chapter 6 for other data structures that support fast DECREASE-KEY operations when the sequence of values returned by EXTRACT-MIN calls are monotonically increasing over time and the data are integers in a specific range.

NP-Completeness

34 NP-Completeness

Almost all the algorithms we have studied thus far have been ***polynomial-time algorithms***: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k. You might wonder whether *all* problems can be solved in polynomial time. The answer is no. For example, there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how much time we allow. There are also problems that can be solved, but not in time $O(n^k)$ for any constant k. Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

The subject of this chapter, however, is an interesting class of problems, called the "NP-complete" problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. This so-called P $\neq$ NP question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

Several NP-complete problems are particularly tantalizing because they seem on the surface to be similar to problems that we know how to solve in polynomial time. In each of the following pairs of problems, one is solvable in polynomial time and the other is NP-complete, but the difference between problems appears to be slight:

Shortest vs. longest simple paths: In Chapter 24, we saw that even with negative edge weights, we can find *shortest* paths from a single source in a directed graph $G = (V, E)$ in $O(VE)$ time. Finding a *longest* simple path between two vertices is difficult, however. Merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

Euler tour vs. hamiltonian cycle: An ***Euler tour*** of a connected, directed graph $G = (V, E)$ is a cycle that traverses each *edge* of G exactly once, although it is allowed to visit each vertex more than once. By Problem 22-3, we can determine whether a graph has an Euler tour in only $O(E)$ time and, in fact, we can find the edges of the Euler tour in $O(E)$ time. A ***hamiltonian cycle*** of a directed graph $G = (V, E)$ is a simple cycle that contains each *vertex* in V. Determining whether a directed graph has a hamiltonian cycle is NP-complete. (Later in this chapter, we shall prove that determining whether an *undirected* graph has a hamiltonian cycle is NP-complete.)

2-CNF satisfiability vs. 3-CNF satisfiability: A boolean formula contains variables whose values are 0 or 1; boolean connectives such as $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT); and parentheses. A boolean formula is ***satisfiable*** if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1. We shall define terms more formally later in this chapter, but informally, a boolean formula is in ***k-conjunctive normal form***, or k-CNF, if it is the AND of clauses of ORs of exactly k variables or their negations. For example, the boolean formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is in 2-CNF. (It has the satisfying assignment $x_1 = 1, x_2 = 0, x_3 = 1$.) Although we can determine in polynomial time whether a 2-CNF formula is satisfiable, we shall see later in this chapter that determining whether a 3-CNF formula is satisfiable is NP-complete.

NP-completeness and the classes P and NP

Throughout this chapter, we shall refer to three classes of problems: P, NP, and NPC, the latter class being the NP-complete problems. We describe them informally here, and we shall define them more formally later on.

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem. Most of the problems examined in previous chapters are in P.

The class NP consists of those problems that are "verifiable" in polynomial time. What do we mean by a problem being verifiable? If we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem. For example, in the hamiltonian-cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$ of $|V|$ vertices. We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3, \ldots, |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well. As another example, for 3-CNF satisfiability, a certificate would be an assignment of values to variables. We could check in polynomial time that this assignment satisfies the boolean formula.

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate. We shall formalize this notion later in this chapter, but for now we can believe that P $\subseteq$ NP. The open question is whether or not P is a proper subset of NP.

Informally, a problem is in the class NPC—and we refer to it as being ***NP-complete***—if it is in NP and is as "hard" as any problem in NP. We shall formally define what it means to be as hard as any problem in NP later in this chapter. In the meantime, we will state without proof that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time algorithm. Most theoretical computer scientists believe that the NP-complete problems are intractable, since given the wide range of NP-complete problems that have been studied to date—without anyone having discovered a polynomial-time solution to any of them—it would be truly astounding if all of them could be solved in polynomial time. Yet, given the effort devoted thus far to proving that NP-complete problems are intractable—without a conclusive outcome—we cannot rule out the possibility that the NP-complete problems are in fact solvable in polynomial time.

To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness. If you can establish a problem as NP-complete, you provide good evidence for its intractability. As an engineer, you would then do better to spend your time developing an approximation algorithm (see Chapter 35) or solving a tractable special case, rather than searching for a fast algorithm that solves the problem exactly. Moreover, many natural and interesting problems that on the surface seem no harder than sorting, graph searching, or network flow are in fact NP-complete. Therefore, you should become familiar with this remarkable class of problems.

Overview of showing problems to be NP-complete

The techniques we use to show that a particular problem is NP-complete differ fundamentally from the techniques used throughout most of this book to design and analyze algorithms. When we demonstrate that a problem is NP-complete, we are making a statement about how hard it is (or at least how hard we think it is), rather than about how easy it is. We are not trying to prove the existence of an efficient algorithm, but instead that no efficient algorithm is likely to exist. In this way, NP-completeness proofs bear some similarity to the proof in Section 8.1 of an $\Omega(n \lg n)$-time lower bound for any comparison sort algorithm; the specific techniques used for showing NP-completeness differ from the decision-tree method used in Section 8.1, however.

We rely on three key concepts in showing a problem to be NP-complete:

Decision problems vs. optimization problems

Many problems of interest are ***optimization problems***, in which each feasible (i.e., "legal") solution has an associated value, and we wish to find a feasible solution with the best value. For example, in a problem that we call SHORTEST-PATH, we are given an undirected graph G and vertices u and v, and we wish to find a path from u to v that uses the fewest edges. In other words, SHORTEST-PATH is the single-pair shortest-path problem in an unweighted, undirected graph. NP-completeness applies directly not to optimization problems, however, but to ***decision problems***, in which the answer is simply "yes" or "no" (or, more formally, "1" or "0").

Although NP-complete problems are confined to the realm of decision problems, we can take advantage of a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For example, a decision problem related to SHORTEST-PATH is PATH: given a directed graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

The relationship between an optimization problem and its related decision problem works in our favor when we try to show that the optimization problem is "hard." That is because the decision problem is in a sense "easier," or at least "no harder." As a specific example, we can solve PATH by solving SHORTEST-PATH and then comparing the number of edges in the shortest path found to the value of the decision-problem parameter k. In other words, if an optimization problem is easy, its related decision problem is easy as well. Stated in a way that has more relevance to NP-completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard. Thus, even though it restricts attention to decision problems, the theory of NP-completeness often has implications for optimization problems as well.

Reductions

The above notion of showing that one problem is no harder or no easier than another applies even when both problems are decision problems. We take advantage of this idea in almost every NP-completeness proof, as follows. Let us consider a decision problem A, which we would like to solve in polynomial time. We call the input to a particular problem an ***instance*** of that problem; for example, in PATH, an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k. Now suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."

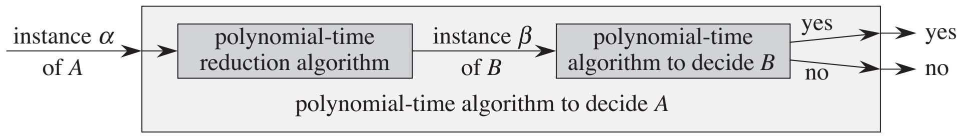

instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A

Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, we transform an instance α of A into an instance β of B, we solve B in polynomial time, and we use the answer for β as the answer for α.

We call such a procedure a polynomial-time ***reduction algorithm*** and, as Figure 34.1 shows, it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B, we use the "easiness" of B to prove the "easiness" of A.

Recalling that NP-completeness is about showing how hard a problem is rather than how easy it is, we use polynomial-time reductions in the opposite way to show that a problem is NP-complete. Let us take the idea a step further, and show how we could use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B. Suppose we have a decision problem A for which we already know that no polynomial-time algorithm can exist. (Let us not concern ourselves for now with how to find such a problem A.) Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B. Now we can use a simple proof by contradiction to show that no polynomial-time algorithm can exist for B. Suppose otherwise; i.e., suppose that B has a polynomial-time algorithm. Then, using the method shown in Figure 34.1, we would have a way to solve problem A in polynomial time, which contradicts our assumption that there is no polynomial-time algorithm for A.

For NP-completeness, we cannot assume that there is absolutely no polynomial-time algorithm for problem A. The proof methodology is similar, however, in that we prove that problem B is NP-complete on the assumption that problem A is also NP-complete.

A first NP-complete problem

Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a "first" NP-complete problem. The problem we shall use is the circuit-satisfiability problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of boolean inputs to this circuit that causes its output to be 1. We shall prove that this first problem is NP-complete in Section 34.3.

Chapter outline

This chapter studies the aspects of NP-completeness that bear most directly on the analysis of algorithms. In Section 34.1, we formalize our notion of "problem" and define the complexity class P of polynomial-time solvable decision problems. We also see how these notions fit into the framework of formal-language theory. Section 34.2 defines the class NP of decision problems whose solutions are verifiable in polynomial time. It also formally poses the P $\neq$ NP question.

Section 34.3 shows we can relate problems via polynomial-time "reductions." It defines NP-completeness and sketches a proof that one problem, called "circuit satisfiability," is NP-complete. Having found one NP-complete problem, we show in Section 34.4 how to prove other problems to be NP-complete much more simply by the methodology of reductions. We illustrate this methodology by showing that two formula-satisfiability problems are NP-complete. With additional reductions, we show in Section 34.5 a variety of other problems to be NP-complete.

34.1 Polynomial time

We begin our study of NP-completeness by formalizing our notion of polynomial-time solvable problems. We generally regard these problems as tractable, but for philosophical, not mathematical, reasons. We can offer three supporting arguments.

First, although we may reasonably regard a problem that requires time $\Theta(n^{100})$ to be intractable, very few practical problems require time on the order of such a high-degree polynomial. The polynomial-time computable problems encountered in practice typically require much less time. Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow. Even if the current best algorithm for a problem has a running time of $\Theta(n^{100})$, an algorithm with a much better running time will likely soon be discovered.

Second, for many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another. For example, the class of problems solvable in polynomial time by the serial random-access machine used throughout most of this book is the same as the class of problems solvable in polynomial time on abstract Turing machines.[1] It is also the same as the class of problems solvable in polynomial time on a parallel computer when the number of processors grows polynomially with the input size.

Third, the class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition. For example, if the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial. Exercise 34.1-5 asks you to show that if an algorithm makes a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then the running time of the composite algorithm is polynomial.

Abstract problems

To understand the class of polynomial-time solvable problems, we must first have a formal notion of what a "problem" is. We define an ***abstract problem*** Q to be a binary relation on a set I of problem ***instances*** and a set S of problem ***solutions***. For example, an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices. A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists. The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices. Since shortest paths are not necessarily unique, a given problem instance may have more than one solution.

This formulation of an abstract problem is more general than we need for our purposes. As we saw above, the theory of NP-completeness restricts attention to ***decision problems***: those having a yes/no solution. In this case, we can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$. For example, a decision problem related to SHORTEST-PATH is the problem PATH that we saw earlier. If $i = \langle G, u, v, k \rangle$ is an instance of the decision problem PATH, then PATH$(i) = 1$ (yes) if a shortest path from u to v has at most k edges, and PATH$(i) = 0$ (no) otherwise. Many abstract problems are not decision problems, but rather ***optimization problems***, which require some value to be minimized or maximized. As we saw above, however, we can usually recast an optimization problem as a decision problem that is no harder.

[1] See Hopcroft and Ullman [180] or Lewis and Papadimitriou [236] for a thorough treatment of the Turing-machine model.

Encodings

In order for a computer program to solve an abstract problem, we must represent problem instances in a way that the program understands. An ***encoding*** of a set S of abstract objects is a mapping e from S to the set of binary strings.[2] For example, we are all familiar with encoding the natural numbers $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ as the strings $\{0, 1, 10, 11, 100, \ldots\}$. Using this encoding, $e(17) = 10001$. If you have looked at computer representations of keyboard characters, you probably have seen the ASCII code, where, for example, the encoding of A is 1000001. We can encode a compound object as a binary string by combining the representations of its constituent parts. Polygons, graphs, functions, ordered pairs, programs—all can be encoded as binary strings.

Thus, a computer algorithm that "solves" some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a ***concrete problem***. We say that an algorithm ***solves*** a concrete problem in time $O(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$ time.[3] A concrete problem is ***polynomial-time solvable***, therefore, if there exists an algorithm to solve it in time $O(n^k)$ for some constant k.

We can now formally define the ***complexity class*** **P** as the set of concrete decision problems that are polynomial-time solvable.

We can use encodings to map abstract problems to concrete problems. Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$, an encoding $e : I \to \{0, 1\}^*$ can induce a related concrete decision problem, which we denote by $e(Q)$.[4] If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0, 1\}$, then the solution to the concrete-problem instance $e(i) \in \{0, 1\}^*$ is also $Q(i)$. As a technicality, some binary strings might represent no meaningful abstract-problem instance. For convenience, we shall assume that any such string maps arbitrarily to 0. Thus, the concrete problem produces the same solutions as the abstract problem on binary-string instances that represent the encodings of abstract-problem instances.

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge, but we would

[2]The codomain of e need not be *binary* strings; any set of strings over a finite alphabet having at least 2 symbols will do.

[3]We assume that the algorithm's output is separate from its input. Because it takes at least one time step to produce each bit of the output and the algorithm takes $O(T(n))$ time steps, the size of the output is $O(T(n))$.

[4]We denote by $\{0, 1\}^*$ the set of all strings composed of symbols from the set $\{0, 1\}$.

like the definition to be independent of any particular encoding. That is, the efficiency of solving a problem should not depend on how the problem is encoded. Unfortunately, it depends quite heavily on the encoding. For example, suppose that an integer k is to be provided as the sole input to an algorithm, and suppose that the running time of the algorithm is $\Theta(k)$. If the integer k is provided in ***unary***—a string of k 1s—then the running time of the algorithm is $O(n)$ on length-n inputs, which is polynomial time. If we use the more natural binary representation of the integer k, however, then the input length is $n = \lfloor \lg k \rfloor + 1$. In this case, the running time of the algorithm is $\Theta(k) = \Theta(2^n)$, which is exponential in the size of the input. Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

How we encode an abstract problem matters quite a bit to how we understand polynomial time. We cannot really talk about solving an abstract problem without first specifying an encoding. Nevertheless, in practice, if we rule out "expensive" encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time. For example, representing integers in base 3 instead of binary has no effect on whether a problem is solvable in polynomial time, since we can convert an integer represented in base 3 to an integer represented in base 2 in polynomial time.

We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is ***polynomial-time computable*** if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$. For some set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomially related*** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.[5] That is, a polynomial-time algorithm can compute the encoding $e_2(i)$ from the encoding $e_1(i)$, and vice versa. If two encodings e_1 and e_2 of an abstract problem are polynomially related, whether the problem is polynomial-time solvable or not is independent of which encoding we use, as the following lemma shows.

Lemma 34.1

Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in \text{P}$ if and only if $e_2(Q) \in \text{P}$.

[5]Technically, we also require the functions f_{12} and f_{21} to "map noninstances to noninstances." A ***noninstance*** of an encoding e is a string $x \in \{0,1\}^*$ such that there is no instance i for which $e(i) = x$. We require that $f_{12}(x) = y$ for every noninstance x of encoding e_1, where y is some noninstance of e_2, and that $f_{21}(x') = y'$ for every noninstance x' of e_2, where y' is some noninstance of e_1.

Proof We need only prove the forward direction, since the backward direction is symmetric. Suppose, therefore, that $e_1(Q)$ can be solved in time $O(n^k)$ for some constant k. Further, suppose that for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in time $O(n^c)$ for some constant c, where $n = |e_2(i)|$. To solve problem $e_2(Q)$, on input $e_2(i)$, we first compute $e_1(i)$ and then run the algorithm for $e_1(Q)$ on $e_1(i)$. How long does this take? Converting encodings takes time $O(n^c)$, and therefore $|e_1(i)| = O(n^c)$, since the output of a serial computer cannot be longer than its running time. Solving the problem on $e_1(i)$ takes time $O(|e_1(i)|^k) = O(n^{ck})$, which is polynomial since both c and k are constants. ■

Thus, whether an abstract problem has its instances encoded in binary or base 3 does not affect its "complexity," that is, whether it is polynomial-time solvable or not; but if instances are encoded in unary, its complexity may change. In order to be able to converse in an encoding-independent fashion, we shall generally assume that problem instances are encoded in any reasonable, concise fashion, unless we specifically say otherwise. To be precise, we shall assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas. (ASCII is one such encoding scheme.) With such a "standard" encoding in hand, we can derive reasonable encodings of other mathematical objects, such as tuples, graphs, and formulas. To denote the standard encoding of an object, we shall enclose the object in angle braces. Thus, $\langle G \rangle$ denotes the standard encoding of a graph G.

As long as we implicitly use an encoding that is polynomially related to this standard encoding, we can talk directly about abstract problems without reference to any particular encoding, knowing that the choice of encoding has no effect on whether the abstract problem is polynomial-time solvable. Henceforth, we shall generally assume that all problem instances are binary strings encoded using the standard encoding, unless we explicitly specify the contrary. We shall also typically neglect the distinction between abstract and concrete problems. You should watch out for problems that arise in practice, however, in which a standard encoding is not obvious and the encoding does make a difference.

A formal-language framework

By focusing on decision problems, we can take advantage of the machinery of formal-language theory. Let's review some definitions from that theory. An ***alphabet*** Σ is a finite set of symbols. A ***language*** L over Σ is any set of strings made up of symbols from Σ. For example, if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ is the language of binary representations of prime numbers. We denote the ***empty string*** by ε, the ***empty language*** by $\emptyset$, and the language of all strings over Σ by Σ^*. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ is the set of all binary strings. Every language L over Σ is a subset of Σ^*.

We can perform a variety of operations on languages. Set-theoretic operations, such as ***union*** and ***intersection***, follow directly from the set-theoretic definitions. We define the ***complement*** of L by $\overline{L} = \Sigma^* - L$. The ***concatenation*** $L_1 L_2$ of two languages L_1 and L_2 is the language

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\} .$$

The ***closure*** or ***Kleene star*** of a language L is the language

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots ,$$

where L^k is the language obtained by concatenating L to itself k times.

From the point of view of language theory, the set of instances for any decision problem Q is simply the set Σ^*, where $\Sigma = \{0, 1\}$. Since Q is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view Q as a language L over $\Sigma = \{0, 1\}$, where

$$L = \{x \in \Sigma^* : Q(x) = 1\} .$$

For example, the decision problem PATH has the corresponding language

$$\begin{aligned}\text{PATH} = \{\langle G, u, v, k\rangle :\ & G = (V, E) \text{ is an undirected graph,}\\ & u, v \in V,\\ & k \geq 0 \text{ is an integer, and}\\ & \text{there exists a path from } u \text{ to } v \text{ in } G\\ & \text{consisting of at most } k \text{ edges}\} .\end{aligned}$$

(Where convenient, we shall sometimes use the same name—PATH in this case—to refer to both a decision problem and its corresponding language.)

The formal-language framework allows us to express concisely the relation between decision problems and algorithms that solve them. We say that an algorithm A ***accepts*** a string $x \in \{0, 1\}^*$ if, given input x, the algorithm's output $A(x)$ is 1. The language ***accepted*** by an algorithm A is the set of strings $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, that is, the set of strings that the algorithm accepts. An algorithm A ***rejects*** a string x if $A(x) = 0$.

Even if language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm may loop forever. A language L is ***decided*** by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A. A language L is ***accepted in polynomial time*** by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$. A language L is ***decided in polynomial time*** by an algorithm A if there exists a constant k such that for any length-n string $x \in \{0,1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$. Thus, to accept a language, an algorithm need only produce an answer when provided a string in L, but to decide a language, it must correctly accept or reject every string in $\{0,1\}^*$.

As an example, the language PATH can be accepted in polynomial time. One polynomial-time accepting algorithm verifies that G encodes an undirected graph, verifies that u and v are vertices in G, uses breadth-first search to compute a shortest path from u to v in G, and then compares the number of edges on the shortest path obtained with k. If G encodes an undirected graph and the path found from u to v has at most k edges, the algorithm outputs 1 and halts. Otherwise, the algorithm runs forever. This algorithm does not decide PATH, however, since it does not explicitly output 0 for instances in which a shortest path has more than k edges. A decision algorithm for PATH must explicitly reject binary strings that do not belong to PATH. For a decision problem such as PATH, such a decision algorithm is easy to design: instead of running forever when there is not a path from u to v with at most k edges, it outputs 0 and halts. (It must also output 0 and halt if the input encoding is faulty.) For other problems, such as Turing's Halting Problem, there exists an accepting algorithm, but no decision algorithm exists.

We can informally define a ***complexity class*** as a set of languages, membership in which is determined by a ***complexity measure***, such as running time, of an algorithm that determines whether a given string x belongs to language L. The actual definition of a complexity class is somewhat more technical.[6]

Using this language-theoretic framework, we can provide an alternative definition of the complexity class P:

$$\text{P} = \{L \subseteq \{0,1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}\ .$$

In fact, P is also the class of languages that can be accepted in polynomial time.

Theorem 34.2
$\text{P} = \{L : L \text{ is accepted by a polynomial-time algorithm}\}$.

Proof Because the class of languages decided by polynomial-time algorithms is a subset of the class of languages accepted by polynomial-time algorithms, we need only show that if L is accepted by a polynomial-time algorithm, it is decided by a polynomial-time algorithm. Let L be the language accepted by some

[6]For more on complexity classes, see the seminal paper by Hartmanis and Stearns [162].

polynomial-time algorithm A. We shall use a classic "simulation" argument to construct another polynomial-time algorithm A' that decides L. Because A accepts L in time $O(n^k)$ for some constant k, there also exists a constant c such that A accepts L in at most cn^k steps. For any input string x, the algorithm A' simulates cn^k steps of A. After simulating cn^k steps, algorithm A' inspects the behavior of A. If A has accepted x, then A' accepts x by outputting a 1. If A has not accepted x, then A' rejects x by outputting a 0. The overhead of A' simulating A does not increase the running time by more than a polynomial factor, and thus A' is a polynomial-time algorithm that decides L. ■

Note that the proof of Theorem 34.2 is nonconstructive. For a given language $L \in \mathrm{P}$, we may not actually know a bound on the running time for the algorithm A that accepts L. Nevertheless, we know that such a bound exists, and therefore, that an algorithm A' exists that can check the bound, even though we may not be able to find the algorithm A' easily.

Exercises

34.1-1

Define the optimization problem LONGEST-PATH-LENGTH as the relation that associates each instance of an undirected graph and two vertices with the number of edges in a longest simple path between the two vertices. Define the decision problem LONGEST-PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ is an undirected graph, $u, v \in V$, $k \geq 0$ is an integer, and there exists a simple path from u to v in G consisting of at least k edges$\}$. Show that the optimization problem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if LONGEST-PATH $\in$ P.

34.1-2

Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.

34.1-3

Give a formal encoding of directed graphs as binary strings using an adjacency-matrix representation. Do the same using an adjacency-list representation. Argue that the two representations are polynomially related.

34.1-4

Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked for in Exercise 16.2-2 a polynomial-time algorithm? Explain your answer.

34.1-5

Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

34.1-6

Show that the class P, viewed as a set of languages, is closed under union, intersection, concatenation, complement, and Kleene star. That is, if $L_1, L_2 \in \mathrm{P}$, then $L_1 \cup L_2 \in \mathrm{P}$, $L_1 \cap L_2 \in \mathrm{P}$, $L_1 L_2 \in \mathrm{P}$, $\overline{L_1} \in \mathrm{P}$, and $L_1^* \in \mathrm{P}$.

34.2 Polynomial-time verification

We now look at algorithms that verify membership in languages. For example, suppose that for a given instance $\langle G, u, v, k \rangle$ of the decision problem PATH, we are also given a path p from u to v. We can easily check whether p is a path in G and whether the length of p is at most k, and if so, we can view p as a "certificate" that the instance indeed belongs to PATH. For the decision problem PATH, this certificate doesn't seem to buy us much. After all, PATH belongs to P—in fact, we can solve PATH in linear time—and so verifying membership from a given certificate takes as long as solving the problem from scratch. We shall now examine a problem for which we know of no polynomial-time decision algorithm and yet, given a certificate, verification is easy.

Hamiltonian cycles

The problem of finding a hamiltonian cycle in an undirected graph has been studied for over a hundred years. Formally, a ***hamiltonian cycle*** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be ***hamiltonian***; otherwise, it is ***nonhamiltonian***. The name honors W. R. Hamilton, who described a mathematical game on the dodecahedron (Figure 34.2(a)) in which one player sticks five pins in any five consecutive vertices and the other player must complete the path to form a cycle

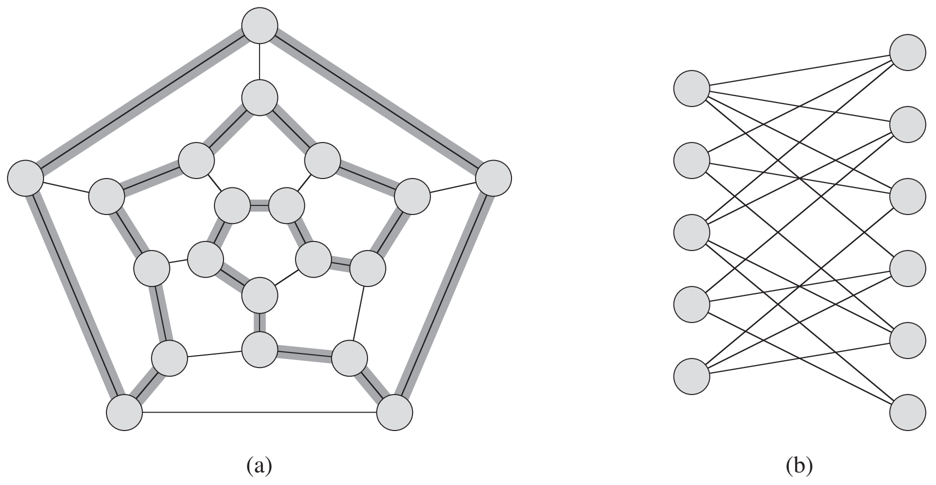


Figure 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

containing all the vertices.[7] The dodecahedron is hamiltonian, and Figure 34.2(a) shows one hamiltonian cycle. Not all graphs are hamiltonian, however. For example, Figure 34.2(b) shows a bipartite graph with an odd number of vertices. Exercise 34.2-2 asks you to show that all such graphs are nonhamiltonian.

We can define the ***hamiltonian-cycle problem***, "Does a graph G have a hamiltonian cycle?" as a formal language:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\} .$$

How might an algorithm decide the language HAM-CYCLE? Given a problem instance $\langle G \rangle$, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a hamiltonian path. What is the running time of this algorithm? If we use the "reasonable" encoding of a graph as its adjacency matrix, the number m of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |\langle G \rangle|$ is the length of the encoding of G. There are $m!$ possible permutations

[7]In a letter dated 17 October 1856 to his friend John T. Graves, Hamilton [157, p. 624] wrote, "I have found that some young persons have been much amused by trying a new mathematical game which the Icosion furnishes, one person sticking five pins in any five consecutive points . . . and the other player then aiming to insert, which by the theory in this letter can always be done, fifteen other pins, in cyclical succession, so as to cover all the other points, and to end in immediate proximity to the pin wherewith his antagonist had begun."

of the vertices, and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$, which is not $O(n^k)$ for any constant k. Thus, this naive algorithm does not run in polynomial time. In fact, the hamiltonian-cycle problem is NP-complete, as we shall prove in Section 34.5.

Verification algorithms

Consider a slightly easier problem. Suppose that a friend tells you that a given graph G is hamiltonian, and then offers to prove it by giving you the vertices in order along the hamiltonian cycle. It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph. You could certainly implement this verification algorithm to run in $O(n^2)$ time, where n is the length of the encoding of G. Thus, a proof that a hamiltonian cycle exists in a graph can be verified in polynomial time.

We define a ***verification algorithm*** as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***. A two-argument algorithm A ***verifies*** an input string x if there exists a certificate y such that $A(x, y) = 1$. The ***language verified*** by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\} .$$

Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$. For example, in the hamiltonian-cycle problem, the certificate is the list of vertices in some hamiltonian cycle. If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify this fact. Conversely, if a graph is not hamiltonian, there can be no list of vertices that fools the verification algorithm into believing that the graph is hamiltonian, since the verification algorithm carefully checks the proposed "cycle" to be sure.

The complexity class NP

The ***complexity class*** **NP** is the class of languages that can be verified by a polynomial-time algorithm.[8] More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

$$\begin{aligned} L = \{x \in \{0,1\}^* : {} & \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \\ & \text{such that } A(x,y) = 1\} \, . \end{aligned}$$

We say that algorithm A ***verifies*** language L ***in polynomial time***.

From our earlier discussion on the hamiltonian-cycle problem, we now see that HAM-CYCLE $\in$ NP. (It is always nice to know that an important set is nonempty.) Moreover, if $L \in$ P, then $L \in$ NP, since if there is a polynomial-time algorithm to decide L, the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in L. Thus, P $\subseteq$ NP.

It is unknown whether P $=$ NP, but most researchers believe that P and NP are not the same class. Intuitively, the class P consists of problems that can be solved quickly. The class NP consists of problems for which a solution can be verified quickly. You may have learned from experience that it is often more difficult to solve a problem from scratch than to verify a clearly presented solution, especially when working under time constraints. Theoretical computer scientists generally believe that this analogy extends to the classes P and NP, and thus that NP includes languages that are not in P.

There is more compelling, though not conclusive, evidence that P $\neq$ NP—the existence of languages that are "NP-complete." We shall study this class in Section 34.3.

Many other fundamental questions beyond the P $\neq$ NP question remain unresolved. Figure 34.3 shows some possible scenarios. Despite much work by many researchers, no one even knows whether the class NP is closed under complement. That is, does $L \in$ NP imply $\overline{L} \in$ NP? We can define the ***complexity class*** **co-NP** as the set of languages L such that $\overline{L} \in$ NP. We can restate the question of whether NP is closed under complement as whether NP $=$ co-NP. Since P is closed under complement (Exercise 34.1-6), it follows from Exercise 34.2-9 that P $\subseteq$ NP $\cap$ co-NP. Once again, however, no one knows whether P $=$ NP $\cap$ co-NP or whether there is some language in NP $\cap$ co-NP $-$ P.

[8] The name "NP" stands for "nondeterministic polynomial time." The class NP was originally studied in the context of nondeterminism, but this book uses the somewhat simpler yet equivalent notion of verification. Hopcroft and Ullman [180] give a good presentation of NP-completeness in terms of nondeterministic models of computation.

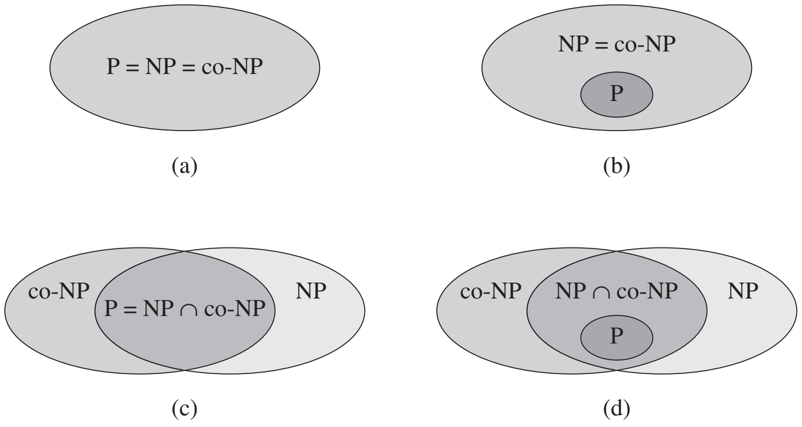


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. **(a)** P $=$ NP $=$ co-NP. Most researchers regard this possibility as the most unlikely. **(b)** If NP is closed under complement, then NP $=$ co-NP, but it need not be the case that P $=$ NP. **(c)** P $=$ NP$\cap$co-NP, but NP is not closed under complement. **(d)** NP $\neq$ co-NP and P $\neq$ NP $\cap$ co-NP. Most researchers regard this possibility as the most likely.

Thus, our understanding of the precise relationship between P and NP is woefully incomplete. Nevertheless, even though we might not be able to prove that a particular problem is intractable, if we can prove that it is NP-complete, then we have gained valuable information about it.

Exercises

34.2-1
Consider the language GRAPH-ISOMORPHISM $= \{\langle G_1, G_2 \rangle : G_1$ and G_2 are isomorphic graphs$\}$. Prove that GRAPH-ISOMORPHISM $\in$ NP by describing a polynomial-time algorithm to verify the language.

34.2-2
Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian.

34.2-3
Show that if HAM-CYCLE $\in$ P, then the problem of listing the vertices of a hamiltonian cycle, in order, is polynomial-time solvable.

34.2-4

Prove that the class NP of languages is closed under union, intersection, concatenation, and Kleene star. Discuss the closure of NP under complement.

34.2-5

Show that any language in NP can be decided by an algorithm running in time $2^{O(n^k)}$ for some constant k.

34.2-6

A ***hamiltonian path*** in a graph is a simple path that visits every vertex exactly once. Show that the language HAM-PATH $= \{\langle G, u, v \rangle :$ there is a hamiltonian path from u to v in graph $G\}$ belongs to NP.

34.2-7

Show that the hamiltonian-path problem from Exercise 34.2-6 can be solved in polynomial time on directed acyclic graphs. Give an efficient algorithm for the problem.

34.2-8

Let ϕ be a boolean formula constructed from the boolean input variables $x_1, x_2, \ldots, x_k$, negations ($\neg$), ANDs ($\wedge$), ORs ($\vee$), and parentheses. The formula ϕ is a ***tautology*** if it evaluates to 1 for every assignment of 1 and 0 to the input variables. Define TAUTOLOGY as the language of boolean formulas that are tautologies. Show that TAUTOLOGY $\in$ co-NP.

34.2-9

Prove that P $\subseteq$ co-NP.

34.2-10

Prove that if NP $\neq$ co-NP, then P $\neq$ NP.

34.2-11

Let G be a connected, undirected graph with at least 3 vertices, and let G^3 be the graph obtained by connecting all pairs of vertices that are connected by a path in G of length at most 3. Prove that G^3 is hamiltonian. (*Hint:* Construct a spanning tree for G, and use an inductive argument.)

34.3 NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe that P $\neq$ NP comes from the existence of the class of "NP-complete" problems. This class has the intriguing property that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time solution, that is, P $=$ NP. Despite years of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem.

The language HAM-CYCLE is one NP-complete problem. If we could decide HAM-CYCLE in polynomial time, then we could solve every problem in NP in polynomial time. In fact, if NP $-$ P should turn out to be nonempty, we could say with certainty that HAM-CYCLE $\in$ NP $-$ P.

The NP-complete languages are, in a sense, the "hardest" languages in NP. In this section, we shall show how to compare the relative "hardness" of languages using a precise notion called "polynomial-time reducibility." Then we formally define the NP-complete languages, and we finish by sketching a proof that one such language, called CIRCUIT-SAT, is NP-complete. In Sections 34.4 and 34.5, we shall use the notion of reducibility to show that many other problems are NP-complete.

Reducibility

Intuitively, a problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q. For example, the problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations. Given an instance $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$. Thus, if a problem Q reduces to another problem Q', then Q is, in a sense, "no harder to solve" than Q'.

Returning to our formal-language framework for decision problems, we say that a language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_{\text{P}} L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \text{ if and only if } f(x) \in L_2 . \tag{34.1}$$

We call the function f the ***reduction function***, and a polynomial-time algorithm F that computes f is a ***reduction algorithm***.

Figure 34.4 illustrates the idea of a polynomial-time reduction from a language L_1 to another language L_2. Each language is a subset of $\{0,1\}^*$. The reduction function f provides a polynomial-time mapping such that if $x \in L_1$,

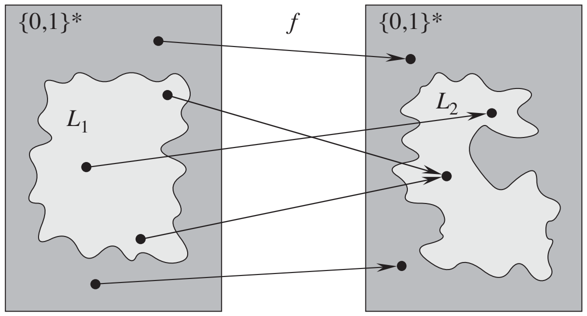


Figure 34.4 An illustration of a polynomial-time reduction from a language L_1 to a language L_2 via a reduction function f. For any input $x \in \{0,1\}^*$, the question of whether $x \in L_1$ has the same answer as the question of whether $f(x) \in L_2$.

then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$. Thus, the reduction function maps any instance x of the decision problem represented by the language L_1 to an instance $f(x)$ of the problem represented by L_2. Providing an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$.

Polynomial-time reductions give us a powerful tool for proving that various languages belong to P.

Lemma 34.3
If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \le_P L_2$, then $L_2 \in P$ implies $L_1 \in P$.

Proof Let A_2 be a polynomial-time algorithm that decides L_2, and let F be a polynomial-time reduction algorithm that computes the reduction function f. We shall construct a polynomial-time algorithm A_1 that decides L_1.

Figure 34.5 illustrates how we construct A_1. For a given input $x \in \{0,1\}^*$, algorithm A_1 uses F to transform x into $f(x)$, and then it uses A_2 to test whether $f(x) \in L_2$. Algorithm A_1 takes the output from algorithm A_2 and produces that answer as its own output.

The correctness of A_1 follows from condition (34.1). The algorithm runs in polynomial time, since both F and A_2 run in polynomial time (see Exercise 34.1-5). ■

NP-completeness

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if $L_1 \le_P L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is

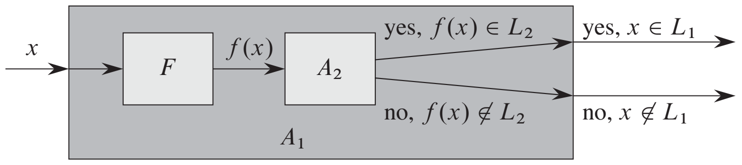


Figure 34.5 The proof of Lemma 34.3. The algorithm F is a reduction algorithm that computes the reduction function f from L_1 to L_2 in polynomial time, and A_2 is a polynomial-time algorithm that decides L_2. Algorithm A_1 decides whether $x \in L_1$ by using F to transform any input x into $f(x)$ and then using A_2 to decide whether $f(x) \in L_2$.

why the "less than or equal to" notation for reduction is mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP.

A language $L \subseteq \{0, 1\}^*$ is ***NP-complete*** if

1. $L \in$ NP, and
2. $L' \leq_P L$ for every $L' \in$ NP.

If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard***. We also define NPC to be the class of NP-complete languages.

As the following theorem shows, NP-completeness is at the crux of deciding whether P is in fact equal to NP.

Theorem 34.4
If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Proof Suppose that $L \in$ P and also that $L \in$ NPC. For any $L' \in$ NP, we have $L' \leq_P L$ by property 2 of the definition of NP-completeness. Thus, by Lemma 34.3, we also have that $L' \in$ P, which proves the first statement of the theorem.

To prove the second statement, note that it is the contrapositive of the first statement. ■

It is for this reason that research into the P $\neq$ NP question centers around the NP-complete problems. Most theoretical computer scientists believe that P $\neq$ NP, which leads to the relationships among P, NP, and NPC shown in Figure 34.6. But, for all we know, someone may yet come up with a polynomial-time algorithm for an NP-complete problem, thus proving that P = NP. Nevertheless, since no polynomial-time algorithm for any NP-complete problem has yet been discov-

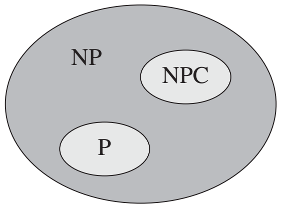


Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $P \cap NPC = \emptyset$.

ered, a proof that a problem is NP-complete provides excellent evidence that it is intractable.

Circuit satisfiability

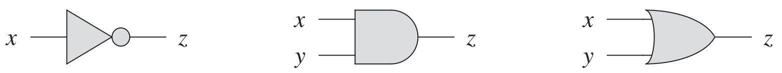
We have defined the notion of an NP-complete problem, but up to this point, we have not actually proved that any problem is NP-complete. Once we prove that at least one problem is NP-complete, we can use polynomial-time reducibility as a tool to prove other problems to be NP-complete. Thus, we now focus on demonstrating the existence of an NP-complete problem: the circuit-satisfiability problem.

Unfortunately, the formal proof that the circuit-satisfiability problem is NP-complete requires technical detail beyond the scope of this text. Instead, we shall informally describe a proof that relies on a basic understanding of boolean combinational circuits.

Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires. A ***boolean combinational element*** is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function. Boolean values are drawn from the set $\{0, 1\}$, where 0 represents FALSE and 1 represents TRUE.

The boolean combinational elements that we use in the circuit-satisfiability problem compute simple boolean functions, and they are known as ***logic gates***. Figure 34.7 shows the three basic logic gates that we use in the circuit-satisfiability problem: the ***NOT gate*** (or ***inverter***), the ***AND gate***, and the ***OR gate***. The NOT gate takes a single binary ***input*** x, whose value is either 0 or 1, and produces a binary ***output*** z whose value is opposite that of the input value. Each of the other two gates takes two binary inputs x and y and produces a single binary output z.

We can describe the operation of each gate, and of any boolean combinational element, by a ***truth table***, shown under each gate in Figure 34.7. A truth table gives the outputs of the combinational element for each possible setting of the inputs. For



x	$\neg x$
0	1
1	0

(a)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(c)

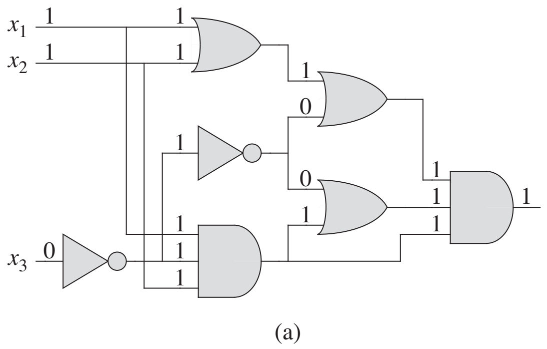
Figure 34.7 Three basic logic gates, with binary inputs and outputs. Under each gate is the truth table that describes the gate's operation. **(a)** The NOT gate. **(b)** The AND gate. **(c)** The OR gate.

example, the truth table for the OR gate tells us that when the inputs are $x = 0$ and $y = 1$, the output value is $z = 1$. We use the symbols $\neg$ to denote the NOT function, $\wedge$ to denote the AND function, and $\vee$ to denote the OR function. Thus, for example, $0 \vee 1 = 1$.

We can generalize AND and OR gates to take more than two inputs. An AND gate's output is 1 if all of its inputs are 1, and its output is 0 otherwise. An OR gate's output is 1 if any of its inputs are 1, and its output is 0 otherwise.

A ***boolean combinational circuit*** consists of one or more boolean combinational elements interconnected by ***wires***. A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second. Figure 34.8 shows two similar boolean combinational circuits, differing in only one gate. Part (a) of the figure also shows the values on the individual wires, given the input $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$. Although a single wire may have no more than one combinational-element output connected to it, it can feed several element inputs. The number of element inputs fed by a wire is called the ***fan-out*** of the wire. If no element output is connected to a wire, the wire is a ***circuit input***, accepting input values from an external source. If no element input is connected to a wire, the wire is a ***circuit output***, providing the results of the circuit's computation to the outside world. (An internal wire can also fan out to a circuit output.) For the purpose of defining the circuit-satisfiability problem, we limit the number of circuit outputs to 1, though in actual hardware design, a boolean combinational circuit may have multiple outputs.

Boolean combinational circuits contain no cycles. In other words, suppose we create a directed graph $G = (V, E)$ with one vertex for each combinational element and with k directed edges for each wire whose fan-out is k; the graph contains a directed edge (u, v) if a wire connects the output of element u to an input of element v. Then G must be acyclic.



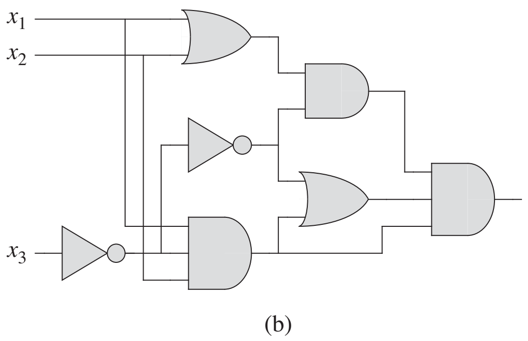


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

A ***truth assignment*** for a boolean combinational circuit is a set of boolean input values. We say that a one-output boolean combinational circuit is ***satisfiable*** if it has a ***satisfying assignment***: a truth assignment that causes the output of the circuit to be 1. For example, the circuit in Figure 34.8(a) has the satisfying assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$, and so it is satisfiable. As Exercise 34.3-1 asks you to show, no assignment of values to x_1, x_2, and x_3 causes the circuit in Figure 34.8(b) to produce a 1 output; it always produces 0, and so it is unsatisfiable.

The ***circuit-satisfiability problem*** is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?" In order to pose this question formally, however, we must agree on a standard encoding for circuits. The ***size*** of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit. We could devise a graphlike encoding that maps any given circuit C into a binary string $\langle C \rangle$ whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

$$\text{CIRCUIT-SAT} = \{\langle C \rangle : C \text{ is a satisfiable boolean combinational circuit}\} .$$

The circuit-satisfiability problem arises in the area of computer-aided hardware optimization. If a subcircuit always produces 0, that subcircuit is unnecessary; the designer can replace it by a simpler subcircuit that omits all logic gates and provides the constant 0 value as its output. You can see why we would like to have a polynomial-time algorithm for this problem.

Given a circuit C, we might attempt to determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if the circuit has k inputs, then we would have to check up to 2^k possible assignments. When the size of C is polynomial in k, checking each one takes $\Omega(2^k)$ time, which is superpolynomial in the size of the circuit.[9] In fact, as we have claimed, there is strong evidence that no polynomial-time algorithm exists that solves the circuit-satisfiability problem because circuit satisfiability is NP-complete. We break the proof of this fact into two parts, based on the two parts of the definition of NP-completeness.

Lemma 34.5
The circuit-satisfiability problem belongs to the class NP.

Proof We shall provide a two-input, polynomial-time algorithm A that can verify CIRCUIT-SAT. One of the inputs to A is (a standard encoding of) a boolean combinational circuit C. The other input is a certificate corresponding to an assignment of boolean values to the wires in C. (See Exercise 34.3-4 for a smaller certificate.)

We construct the algorithm A as follows. For each logic gate in the circuit, it checks that the value provided by the certificate on the output wire is correctly computed as a function of the values on the input wires. Then, if the output of the entire circuit is 1, the algorithm outputs 1, since the values assigned to the inputs of C provide a satisfying assignment. Otherwise, A outputs 0.

Whenever a satisfiable circuit C is input to algorithm A, there exists a certificate whose length is polynomial in the size of C and that causes A to output a 1. Whenever an unsatisfiable circuit is input, no certificate can fool A into believing that the circuit is satisfiable. Algorithm A runs in polynomial time: with a good implementation, linear time suffices. Thus, we can verify CIRCUIT-SAT in polynomial time, and CIRCUIT-SAT $\in$ NP. ■

The second part of proving that CIRCUIT-SAT is NP-complete is to show that the language is NP-hard. That is, we must show that every language in NP is polynomial-time reducible to CIRCUIT-SAT. The actual proof of this fact is full of technical intricacies, and so we shall settle for a sketch of the proof based on some understanding of the workings of computer hardware.

A computer program is stored in the computer memory as a sequence of instructions. A typical instruction encodes an operation to be performed, addresses of operands in memory, and an address where the result is to be stored. A special memory location, called the ***program counter***, keeps track of which instruc-

[9]On the other hand, if the size of the circuit C is $\Theta(2^k)$, then an algorithm whose running time is $O(2^k)$ has a running time that is polynomial in the circuit size. Even if P $\neq$ NP, this situation would not contradict the NP-completeness of the problem; the existence of a polynomial-time algorithm for a special case does not imply that there is a polynomial-time algorithm for all cases.

tion is to be executed next. The program counter automatically increments upon fetching each instruction, thereby causing the computer to execute instructions sequentially. The execution of an instruction can cause a value to be written to the program counter, however, which alters the normal sequential execution and allows the computer to loop and perform conditional branches.

At any point during the execution of a program, the computer's memory holds the entire state of the computation. (We take the memory to include the program itself, the program counter, working storage, and any of the various bits of state that a computer maintains for bookkeeping.) We call any particular state of computer memory a ***configuration***. We can view the execution of an instruction as mapping one configuration to another. The computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit, which we denote by M in the proof of the following lemma.

Lemma 34.6
The circuit-satisfiability problem is NP-hard.

Proof Let L be any language in NP. We shall describe a polynomial-time algorithm F computing a reduction function f that maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT.

Since $L \in$ NP, there must exist an algorithm A that verifies L in polynomial time. The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

Let $T(n)$ denote the worst-case running time of algorithm A on length-n input strings, and let $k \geq 1$ be a constant such that $T(n) = O(n^k)$ and the length of the certificate is $O(n^k)$. (The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate, but since the length of the certificate is polynomial in the length n of the input string, the running time is polynomial in n.)

The basic idea of the proof is to represent the computation of A as a sequence of configurations. As Figure 34.9 illustrates, we can break each configuration into parts consisting of the program for A, the program counter and auxiliary machine state, the input x, the certificate y, and working storage. The combinational circuit M, which implements the computer hardware, maps each configuration c_i to the next configuration c_{i+1}, starting from the initial configuration c_0. Algorithm A writes its output—0 or 1—to some designated location by the time it finishes executing, and if we assume that thereafter A halts, the value never changes. Thus, if the algorithm runs for at most $T(n)$ steps, the output appears as one of the bits in $c_{T(n)}$.

The reduction algorithm F constructs a single combinational circuit that computes all configurations produced by a given initial configuration. The idea is to

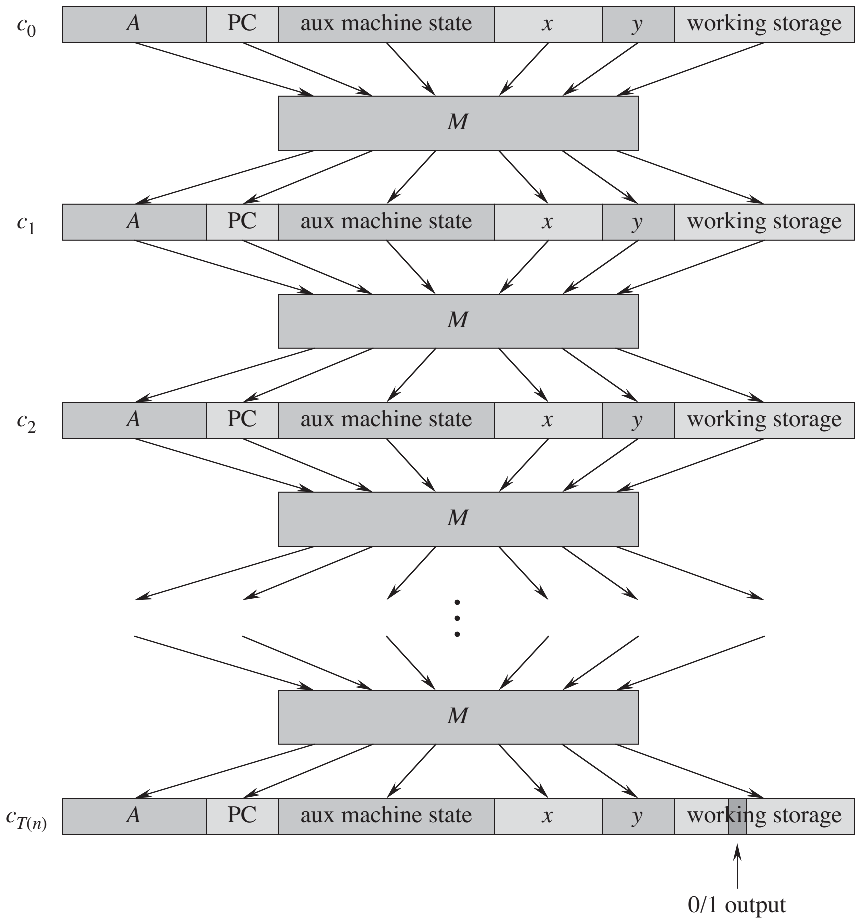


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. A boolean combinational circuit M maps each configuration to the next configuration. The output is a distinguished bit in the working storage.

paste together $T(n)$ copies of the circuit M. The output of the ith circuit, which produces configuration c_i, feeds directly into the input of the $(i+1)$st circuit. Thus, the configurations, rather than being stored in the computer's memory, simply reside as values on the wires connecting copies of M.

Recall what the polynomial-time reduction algorithm F must do. Given an input x, it must compute a circuit $C = f(x)$ that is satisfiable if and only if there exists a certificate y such that $A(x, y) = 1$. When F obtains an input x, it first computes $n = |x|$ and constructs a combinational circuit C' consisting of $T(n)$ copies of M. The input to C' is an initial configuration corresponding to a computation on $A(x, y)$, and the output is the configuration $c_{T(n)}$.

Algorithm F modifies circuit C' slightly to construct the circuit $C = f(x)$. First, it wires the inputs to C' corresponding to the program for A, the initial program counter, the input x, and the initial state of memory directly to these known values. Thus, the only remaining inputs to the circuit correspond to the certificate y. Second, it ignores all outputs from C', except for the one bit of $c_{T(n)}$ corresponding to the output of A. This circuit C, so constructed, computes $C(y) = A(x, y)$ for any input y of length $O(n^k)$. The reduction algorithm F, when provided an input string x, computes such a circuit C and outputs it.

We need to prove two properties. First, we must show that F correctly computes a reduction function f. That is, we must show that C is satisfiable if and only if there exists a certificate y such that $A(x, y) = 1$. Second, we must show that F runs in polynomial time.

To show that F correctly computes a reduction function, let us suppose that there exists a certificate y of length $O(n^k)$ such that $A(x, y) = 1$. Then, if we apply the bits of y to the inputs of C, the output of C is $C(y) = A(x, y) = 1$. Thus, if a certificate exists, then C is satisfiable. For the other direction, suppose that C is satisfiable. Hence, there exists an input y to C such that $C(y) = 1$, from which we conclude that $A(x, y) = 1$. Thus, F correctly computes a reduction function.

To complete the proof sketch, we need only show that F runs in time polynomial in $n = |x|$. The first observation we make is that the number of bits required to represent a configuration is polynomial in n. The program for A itself has constant size, independent of the length of its input x. The length of the input x is n, and the length of the certificate y is $O(n^k)$. Since the algorithm runs for at most $O(n^k)$ steps, the amount of working storage required by A is polynomial in n as well. (We assume that this memory is contiguous; Exercise 34.3-5 asks you to extend the argument to the situation in which the locations accessed by A are scattered across a much larger region of memory and the particular pattern of scattering can differ for each input x.)

The combinational circuit M implementing the computer hardware has size polynomial in the length of a configuration, which is $O(n^k)$; hence, the size of M is polynomial in n. (Most of this circuitry implements the logic of the memory system.) The circuit C consists of at most $t = O(n^k)$ copies of M, and hence it has size polynomial in n. The reduction algorithm F can construct C from x in polynomial time, since each step of the construction takes polynomial time. ■

The language CIRCUIT-SAT is therefore at least as hard as any language in NP, and since it belongs to NP, it is NP-complete.

Theorem 34.7
The circuit-satisfiability problem is NP-complete.

Proof Immediate from Lemmas 34.5 and 34.6 and from the definition of NP-completeness. ■

Exercises

34.3-1
Verify that the circuit in Figure 34.8(b) is unsatisfiable.

34.3-2
Show that the $\leq_{\mathrm{P}}$ relation is a transitive relation on languages. That is, show that if $L_1 \leq_{\mathrm{P}} L_2$ and $L_2 \leq_{\mathrm{P}} L_3$, then $L_1 \leq_{\mathrm{P}} L_3$.

34.3-3
Prove that $L \leq_{\mathrm{P}} \overline{L}$ if and only if $\overline{L} \leq_{\mathrm{P}} L$.

34.3-4
Show that we could have used a satisfying assignment as a certificate in an alternative proof of Lemma 34.5. Which certificate makes for an easier proof?

34.3-5
The proof of Lemma 34.6 assumes that the working storage for algorithm A occupies a contiguous region of polynomial size. Where in the proof do we exploit this assumption? Argue that this assumption does not involve any loss of generality.

34.3-6
A language L is ***complete*** for a language class C with respect to polynomial-time reductions if $L \in C$ and $L' \leq_{\mathrm{P}} L$ for all $L' \in C$. Show that $\emptyset$ and $\{0, 1\}^*$ are the only languages in P that are not complete for P with respect to polynomial-time reductions.

34.3-7
Show that, with respect to polynomial-time reductions (see Exercise 34.3-6), L is complete for NP if and only if $\overline{L}$ is complete for co-NP.

34.3-8
The reduction algorithm F in the proof of Lemma 34.6 constructs the circuit $C = f(x)$ based on knowledge of x, A, and k. Professor Sartre observes that the string x is input to F, but only the existence of A, k, and the constant factor implicit in the $O(n^k)$ running time is known to F (since the language L belongs to NP), not their actual values. Thus, the professor concludes that F can't possibly construct the circuit C and that the language CIRCUIT-SAT is not necessarily NP-hard. Explain the flaw in the professor's reasoning.

34.4 NP-completeness proofs

We proved that the circuit-satisfiability problem is NP-complete by a direct proof that $L \leq_P$ CIRCUIT-SAT for every language $L \in$ NP. In this section, we shall show how to prove that languages are NP-complete without directly reducing *every* language in NP to the given language. We shall illustrate this methodology by proving that various formula-satisfiability problems are NP-complete. Section 34.5 provides many more examples of the methodology.

The following lemma is the basis of our method for showing that a language is NP-complete.

Lemma 34.8
If L is a language such that $L' \leq_P L$ for some $L' \in$ NPC, then L is NP-hard. If, in addition, $L \in$ NP, then $L \in$ NPC.

Proof Since L' is NP-complete, for all $L'' \in$ NP, we have $L'' \leq_P L'$. By supposition, $L' \leq_P L$, and thus by transitivity (Exercise 34.3-2), we have $L'' \leq_P L$, which shows that L is NP-hard. If $L \in$ NP, we also have $L \in$ NPC. ■

In other words, by reducing a known NP-complete language L' to L, we implicitly reduce every language in NP to L. Thus, Lemma 34.8 gives us a method for proving that a language L is NP-complete:

1. Prove $L \in$ NP.
2. Select a known NP-complete language L'.

3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
5. Prove that the algorithm computing f runs in polynomial time.

(Steps 2–5 show that L is NP-hard.) This methodology of reducing from a single known NP-complete language is far simpler than the more complicated process of showing directly how to reduce from every language in NP. Proving CIRCUIT-SAT $\in$ NPC has given us a "foot in the door." Because we know that the circuit-satisfiability problem is NP-complete, we now can prove much more easily that other problems are NP-complete. Moreover, as we develop a catalog of known NP-complete problems, we will have more and more choices for languages from which to reduce.

Formula satisfiability

We illustrate the reduction methodology by giving an NP-completeness proof for the problem of determining whether a boolean formula, not a circuit, is satisfiable. This problem has the historical honor of being the first problem ever shown to be NP-complete.

We formulate the ***(formula) satisfiability*** problem in terms of the language SAT as follows. An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables: $x_1, x_2, \ldots, x_n$;
2. m boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if); and
3. parentheses. (Without loss of generality, we assume that there are no redundant parentheses, i.e., a formula contains at most one pair of parentheses per boolean connective.)

We can easily encode a boolean formula ϕ in a length that is polynomial in $n + m$. As in boolean combinational circuits, a ***truth assignment*** for a boolean formula ϕ is a set of values for the variables of ϕ, and a ***satisfying assignment*** is a truth assignment that causes it to evaluate to 1. A formula with a satisfying assignment is a ***satisfiable*** formula. The satisfiability problem asks whether a given boolean formula is satisfiable; in formal-language terms,

$$\text{SAT} = \{\langle \phi \rangle : \phi \text{ is a satisfiable boolean formula}\} .$$

As an example, the formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$, since

$$\begin{aligned} \phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \qquad (34.2) \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1 , \end{aligned}$$

and thus this formula ϕ belongs to SAT.

The naive algorithm to determine whether an arbitrary boolean formula is satisfiable does not run in polynomial time. A formula with n variables has 2^n possible assignments. If the length of $\langle \phi \rangle$ is polynomial in n, then checking every assignment requires $\Omega(2^n)$ time, which is superpolynomial in the length of $\langle \phi \rangle$. As the following theorem shows, a polynomial-time algorithm is unlikely to exist.

Theorem 34.9
Satisfiability of boolean formulas is NP-complete.

Proof We start by arguing that SAT $\in$ NP. Then we prove that SAT is NP-hard by showing that CIRCUIT-SAT $\leq_P$ SAT; by Lemma 34.8, this will prove the theorem.

To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression, much as we did in equation (34.2) above. This task is easy to do in polynomial time. If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable. Thus, the first condition of Lemma 34.8 for NP-completeness holds.

To prove that SAT is NP-hard, we show that CIRCUIT-SAT $\leq_P$ SAT. In other words, we need to show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time. We can use induction to express any boolean combinational circuit as a boolean formula. We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas. We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Unfortunately, this straightforward method does not amount to a polynomial-time reduction. As Exercise 34.4-1 asks you to show, shared subformulas—which arise from gates whose output wires have fan-out of 2 or more—can cause the size of the generated formula to grow exponentially. Thus, the reduction algorithm must be somewhat more clever.

Figure 34.10 illustrates how we overcome this problem, using as an example the circuit from Figure 34.8(a). For each wire x_i in the circuit C, the formula ϕ

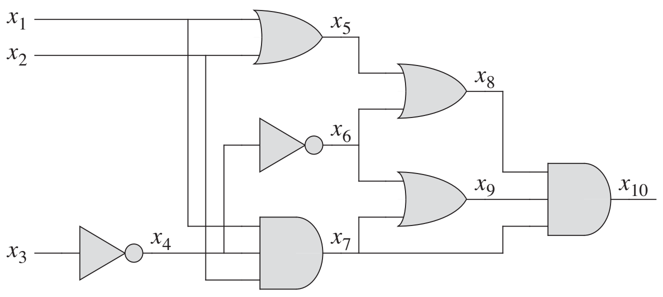


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

has a variable x_i. We can now express how each gate operates as a small formula involving the variables of its incident wires. For example, the operation of the output AND gate is $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$. We call each of these small formulas a ***clause***.

The formula ϕ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate. For the circuit in the figure, the formula is

$$\begin{aligned}
\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3) \\
&\wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
&\wedge (x_6 \leftrightarrow \neg x_4) \\
&\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
&\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
&\wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
&\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) .
\end{aligned}$$

Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.

Why is the circuit C satisfiable exactly when the formula ϕ is satisfiable? If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1. Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument. Thus, we have shown that CIRCUIT-SAT $\leq_P$ SAT, which completes the proof. ■

3-CNF satisfiability

We can prove many problems NP-complete by reducing from formula satisfiability. The reduction algorithm must handle any input formula, though, and this requirement can lead to a huge number of cases that we must consider. We often prefer to reduce from a restricted language of boolean formulas, so that we need to consider fewer cases. Of course, we must not restrict the language so much that it becomes polynomial-time solvable. One convenient language is 3-CNF satisfiability, or 3-CNF-SAT.

We define 3-CNF satisfiability using the following terms. A ***literal*** in a boolean formula is an occurrence of a variable or its negation. A boolean formula is in ***conjunctive normal form***, or ***CNF***, if it is expressed as an AND of ***clauses***, each of which is the OR of one or more literals. A boolean formula is in ***3-conjunctive normal form***, or ***3-CNF***, if each clause has exactly three distinct literals.

For example, the boolean formula

$$(x_1 \lor \neg x_1 \lor \neg x_2) \land (x_3 \lor x_2 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor \neg x_4)$$

is in 3-CNF. The first of its three clauses is $(x_1 \lor \neg x_1 \lor \neg x_2)$, which contains the three literals x_1, $\neg x_1$, and $\neg x_2$.

In 3-CNF-SAT, we are asked whether a given boolean formula ϕ in 3-CNF is satisfiable. The following theorem shows that a polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist, even when they are expressed in this simple normal form.

Theorem 34.10
Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Proof The argument we used in the proof of Theorem 34.9 to show that SAT $\in$ NP applies equally well here to show that 3-CNF-SAT $\in$ NP. By Lemma 34.8, therefore, we need only show that SAT $\leq_P$ 3-CNF-SAT.

We break the reduction algorithm into three basic steps. Each step progressively transforms the input formula ϕ closer to the desired 3-conjunctive normal form.

The first step is similar to the one used to prove CIRCUIT-SAT $\leq_P$ SAT in Theorem 34.9. First, we construct a binary "parse" tree for the input formula ϕ, with literals as leaves and connectives as internal nodes. Figure 34.11 shows such a parse tree for the formula

$$\phi = ((x_1 \to x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2 . \tag{34.3}$$

Should the input formula contain a clause such as the OR of several literals, we use associativity to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. We can now think of the binary parse tree as a circuit for computing the function.

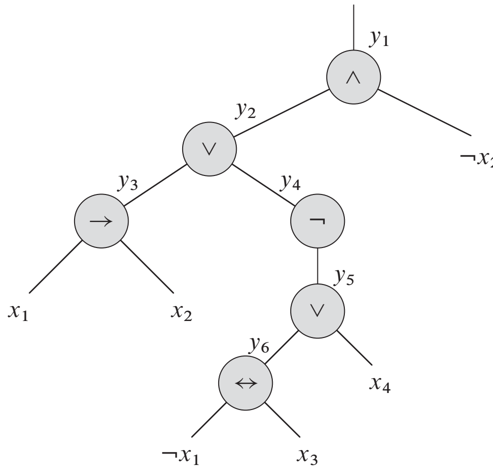


Figure 34.11 The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Mimicking the reduction in the proof of Theorem 34.9, we introduce a variable y_i for the output of each internal node. Then, we rewrite the original formula ϕ as the AND of the root variable and a conjunction of clauses describing the operation of each node. For the formula (34.3), the resulting expression is

$$\begin{aligned} \phi' = y_1 & \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \; . \end{aligned}$$

Observe that the formula ϕ' thus obtained is a conjunction of clauses ϕ'_i, each of which has at most 3 literals. The only requirement that we might fail to meet is that each clause has to be an OR of 3 literals.

The second step of the reduction converts each clause ϕ'_i into conjunctive normal form. We construct a truth table for ϕ'_i by evaluating all possible assignments to its variables. Each row of the truth table consists of a possible assignment of the variables of the clause, together with the value of the clause under that assignment. Using the truth-table entries that evaluate to 0, we build a formula in ***disjunctive normal form*** (or ***DNF***)—an OR of ANDs—that is equivalent to $\neg\phi'_i$. We then negate this formula and convert it into a CNF formula ϕ''_i by using ***DeMorgan's***

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Figure 34.12 The truth table for the clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

laws for propositional logic,

$$\begin{aligned}\neg(a \wedge b) &= \neg a \vee \neg b \,, \\ \neg(a \vee b) &= \neg a \wedge \neg b \,,\end{aligned}$$

to complement all literals, change ORs into ANDs, and change ANDs into ORs.

In our example, we convert the clause $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ into CNF as follows. The truth table for ϕ'_1 appears in Figure 34.12. The DNF formula equivalent to $\neg\phi'_1$ is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2) \,.$$

Negating and applying DeMorgan's laws, we get the CNF formula

$$\begin{aligned}\phi''_1 = {} & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) \,,\end{aligned}$$

which is equivalent to the original clause ϕ'_1.

At this point, we have converted each clause ϕ'_i of the formula ϕ' into a CNF formula ϕ''_i, and thus ϕ' is equivalent to the CNF formula ϕ'' consisting of the conjunction of the ϕ''_i. Moreover, each clause of ϕ'' has at most 3 literals.

The third and final step of the reduction further transforms the formula so that each clause has *exactly* 3 distinct literals. We construct the final 3-CNF formula ϕ''' from the clauses of the CNF formula ϕ''. The formula ϕ''' also uses two auxiliary variables that we shall call p and q. For each clause C_i of ϕ'', we include the following clauses in ϕ''':

- If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ'''.
- If C_i has 2 distinct literals, that is, if $C_i = (l_1 \vee l_2)$, where l_1 and l_2 are literals, then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of ϕ'''. The literals p and $\neg p$ merely fulfill the syntactic requirement that each clause of ϕ''' has

exactly 3 distinct literals. Whether $p = 0$ or $p = 1$, one of the clauses is equivalent to $l_1 \vee l_2$, and the other evaluates to 1, which is the identity for AND.

- If C_i has just 1 distinct literal l, then include $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ as clauses of ϕ'''. Regardless of the values of p and q, one of the four clauses is equivalent to l, and the other 3 evaluate to 1.

We can see that the 3-CNF formula ϕ''' is satisfiable if and only if ϕ is satisfiable by inspecting each of the three steps. Like the reduction from CIRCUIT-SAT to SAT, the construction of ϕ' from ϕ in the first step preserves satisfiability. The second step produces a CNF formula ϕ'' that is algebraically equivalent to ϕ'. The third step produces a 3-CNF formula ϕ''' that is effectively equivalent to ϕ'', since any assignment to the variables p and q produces a formula that is algebraically equivalent to ϕ''.

We must also show that the reduction can be computed in polynomial time. Constructing ϕ' from ϕ introduces at most 1 variable and 1 clause per connective in ϕ. Constructing ϕ'' from ϕ' can introduce at most 8 clauses into ϕ'' for each clause from ϕ', since each clause of ϕ' has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows. The construction of ϕ''' from ϕ'' introduces at most 4 clauses into ϕ''' for each clause of ϕ''. Thus, the size of the resulting formula ϕ''' is polynomial in the length of the original formula. Each of the constructions can easily be accomplished in polynomial time. ■

Exercises

34.4-1

Consider the straightforward (nonpolynomial-time) reduction in the proof of Theorem 34.9. Describe a circuit of size n that, when converted to a formula by this method, yields a formula whose size is exponential in n.

34.4-2

Show the 3-CNF formula that results when we use the method of Theorem 34.10 on the formula (34.3).

34.4-3

Professor Jagger proposes to show that SAT $\leq_P$ 3-CNF-SAT by using only the truth-table technique in the proof of Theorem 34.10, and not the other steps. That is, the professor proposes to take the boolean formula ϕ, form a truth table for its variables, derive from the truth table a formula in 3-DNF that is equivalent to $\neg\phi$, and then negate and apply DeMorgan's laws to produce a 3-CNF formula equivalent to ϕ. Show that this strategy does not yield a polynomial-time reduction.

34.4-4
Show that the problem of determining whether a boolean formula is a tautology is complete for co-NP. (*Hint:* See Exercise 34.3-7.)

34.4-5
Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.

34.4-6
Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

34.4-7
Let 2-CNF-SAT be the set of satisfiable boolean formulas in CNF with exactly 2 literals per clause. Show that 2-CNF-SAT $\in$ P. Make your algorithm as efficient as possible. (*Hint:* Observe that $x \vee y$ is equivalent to $\neg x \rightarrow y$. Reduce 2-CNF-SAT to an efficiently solvable problem on a directed graph.)

34.5 NP-complete problems

NP-complete problems arise in diverse domains: boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, automata and language theory, program optimization, biology, chemistry, physics, and more. In this section, we shall use the reduction methodology to provide NP-completeness proofs for a variety of problems drawn from graph theory and set partitioning.

Figure 34.13 outlines the structure of the NP-completeness proofs in this section and Section 34.4. We prove each language in the figure to be NP-complete by reduction from the language that points to it. At the root is CIRCUIT-SAT, which we proved NP-complete in Theorem 34.7.

34.5.1 The clique problem

A ***clique*** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The ***size*** of a clique is the number of vertices it contains. The ***clique problem*** is the optimization problem of finding a clique of maximum size in

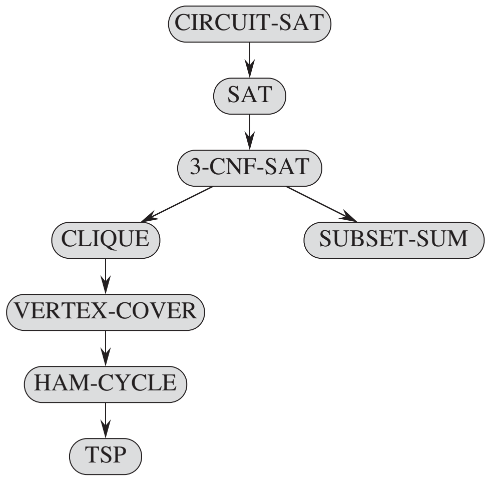


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

a graph. As a decision problem, we ask simply whether a clique of a given size k exists in the graph. The formal definition is

$$\text{CLIQUE} = \{\langle G, k \rangle : G \text{ is a graph containing a clique of size } k\} .$$

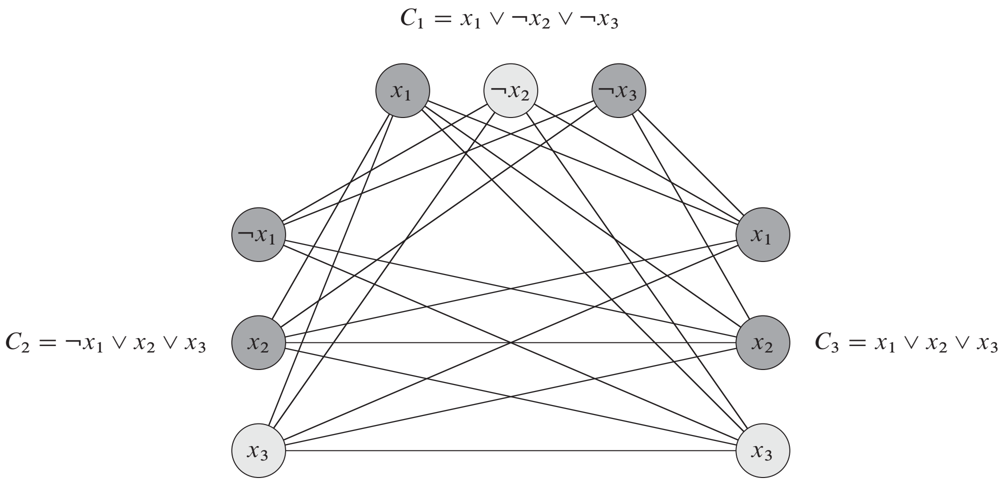
A naive algorithm for determining whether a graph $G = (V, E)$ with $|V|$ vertices has a clique of size k is to list all k-subsets of V, and check each one to see whether it forms a clique. The running time of this algorithm is $\Omega(k^2\binom{|V|}{k})$, which is polynomial if k is a constant. In general, however, k could be near $|V|/2$, in which case the algorithm runs in superpolynomial time. Indeed, an efficient algorithm for the clique problem is unlikely to exist.

Theorem 34.11
The clique problem is NP-complete.

Proof To show that CLIQUE $\in$ NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

We next prove that 3-CNF-SAT $\leq_{\text{P}}$ CLIQUE, which shows that the clique problem is NP-hard. You might be surprised that we should be able to prove such a result, since on the surface logical formulas seem to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses. For $r =$



$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_2
x_3
$C_2 = \neg x_1 \vee x_2 \vee x_3$
x_1
x_2
x_3
$C_3 = x_1 \vee x_2 \vee x_3$

Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

$1, 2, \ldots, k$, each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r. We shall construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k.

We construct the graph $G = (V, E)$ as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, we place a triple of vertices v_1^r, v_2^r, and v_3^r into V. We put an edge between two vertices v_i^r and v_j^s if both of the following hold:

- v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
- their corresponding literals are ***consistent***, that is, l_i^r is not the negation of l_j^s.

We can easily build this graph from ϕ in polynomial time. As an example of this construction, if we have

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) ,$$

then G is the graph shown in Figure 34.14.

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s map to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so ϕ is satisfied. (Any variables that do not correspond to a vertex in the clique may be set arbitrarily.) ■

In the example of Figure 34.14, a satisfying assignment of ϕ has $x_2 = 0$ and $x_3 = 1$. A corresponding clique of size $k = 3$ consists of the vertices corresponding to $\neg x_2$ from the first clause, x_3 from the second clause, and x_3 from the third clause. Because the clique contains no vertices corresponding to either x_1 or $\neg x_1$, we can set x_1 to either 0 or 1 in this satisfying assignment.

Observe that in the proof of Theorem 34.11, we reduced an arbitrary instance of 3-CNF-SAT to an instance of CLIQUE with a particular structure. You might think that we have shown only that CLIQUE is NP-hard in graphs in which the vertices are restricted to occur in triples and in which there are no edges between vertices in the same triple. Indeed, we have shown that CLIQUE is NP-hard only in this restricted case, but this proof suffices to show that CLIQUE is NP-hard in general graphs. Why? If we had a polynomial-time algorithm that solved CLIQUE on general graphs, it would also solve CLIQUE on restricted graphs.

The opposite approach—reducing instances of 3-CNF-SAT with a special structure to general instances of CLIQUE—would not have sufficed, however. Why not? Perhaps the instances of 3-CNF-SAT that we chose to reduce from were "easy," and so we would not have reduced an NP-hard problem to CLIQUE.

Observe also that the reduction used the instance of 3-CNF-SAT, but not the solution. We would have erred if the polynomial-time reduction had relied on knowing whether the formula ϕ is satisfiable, since we do not know how to decide whether ϕ is satisfiable in polynomial time.

34.5.2 The vertex-cover problem

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The ***size*** of a vertex cover is the number of vertices in it. For example, the graph in Figure 34.15(b) has a vertex cover $\{w, z\}$ of size 2.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to

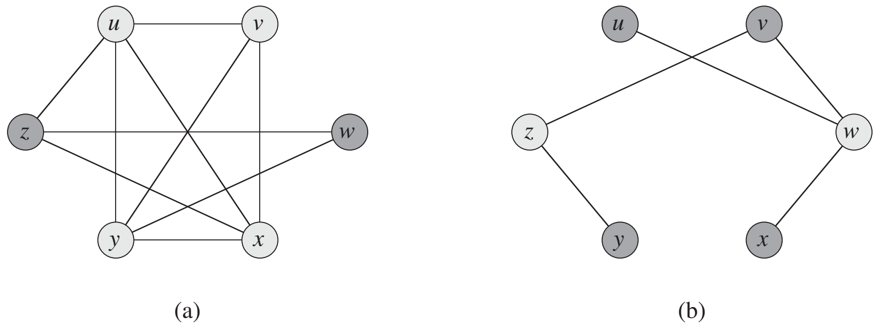


Figure 34.15 Reducing CLIQUE to VERTEX-COVER. **(a)** An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$. **(b)** The graph $\overline{G}$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$.

determine whether a graph has a vertex cover of a given size k. As a language, we define

$$\text{VERTEX-COVER} = \{\langle G, k \rangle : \text{graph } G \text{ has a vertex cover of size } k\} .$$

The following theorem shows that this problem is NP-complete.

Theorem 34.12
The vertex-cover problem is NP-complete.

Proof We first show that VERTEX-COVER $\in$ NP. Suppose we are given a graph $G = (V, E)$ and an integer k. The certificate we choose is the vertex cover $V' \subseteq V$ itself. The verification algorithm affirms that $|V'| = k$, and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. We can easily verify the certificate in polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_{\text{P}}$ VERTEX-COVER. This reduction relies on the notion of the "complement" of a graph. Given an undirected graph $G = (V, E)$, we define the ***complement*** of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$. In other words, $\overline{G}$ is the graph containing exactly those edges that are not in G. Figure 34.15 shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

The reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem. It computes the complement $\overline{G}$, which we can easily do in polynomial time. The output of the reduction algorithm is the instance $\langle \overline{G}, |V| - k \rangle$ of the vertex-cover problem. To complete the proof, we show that this transformation is indeed a reduction: the graph G has a clique of size k if and only if the graph $\overline{G}$ has a vertex cover of size $|V| - k$.

Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\overline{G}$. Let (u, v) be any edge in $\overline{E}$. Then, $(u, v) \notin E$, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, at least one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from $\overline{E}$, every edge of $\overline{E}$ is covered by a vertex in $V - V'$. Hence, the set $V - V'$, which has size $|V| - k$, forms a vertex cover for $\overline{G}$.

Conversely, suppose that $\overline{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then, for all $u, v \in V$, if $(u, v) \in \overline{E}$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words, $V - V'$ is a clique, and it has size $|V| - |V'| = k$. ■

Since VERTEX-COVER is NP-complete, we don't expect to find a polynomial-time algorithm for finding a minimum-size vertex cover. Section 35.1 presents a polynomial-time "approximation algorithm," however, which produces "approximate" solutions for the vertex-cover problem. The size of a vertex cover produced by the algorithm is at most twice the minimum size of a vertex cover.

Thus, we shouldn't give up hope just because a problem is NP-complete. We may be able to design a polynomial-time approximation algorithm that obtains near-optimal solutions, even though finding an optimal solution is NP-complete. Chapter 35 gives several approximation algorithms for NP-complete problems.

34.5.3 The hamiltonian-cycle problem

We now return to the hamiltonian-cycle problem defined in Section 34.2.

Theorem 34.13
The hamiltonian cycle problem is NP-complete.

Proof We first show that HAM-CYCLE belongs to NP. Given a graph $G = (V, E)$, our certificate is the sequence of $|V|$ vertices that makes up the hamiltonian cycle. The verification algorithm checks that this sequence contains each vertex in V exactly once and that with the first vertex repeated at the end, it forms a cycle in G. That is, it checks that there is an edge between each pair of consecutive vertices and between the first and last vertices. We can verify the certificate in polynomial time.

We now prove that VERTEX-COVER $\leq_P$ HAM-CYCLE, which shows that HAM-CYCLE is NP-complete. Given an undirected graph $G = (V, E)$ and an

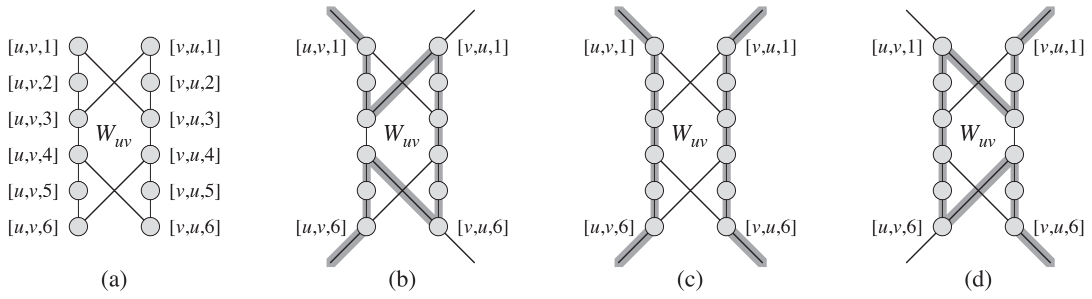


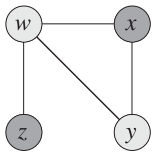
Figure 34.16 The widget used in reducing the vertex-cover problem to the hamiltonian-cycle problem. An edge (u, v) of graph G corresponds to widget W_{uv} in the graph G' created in the reduction. **(a)** The widget, with individual vertices labeled. **(b)–(d)** The shaded paths are the only possible ones through the widget that include all vertices, assuming that the only connections from the widget to the remainder of G' are through vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$.

integer k, we construct an undirected graph $G' = (V', E')$ that has a hamiltonian cycle if and only if G has a vertex cover of size k.

Our construction uses a ***widget***, which is a piece of a graph that enforces certain properties. Figure 34.16(a) shows the widget we use. For each edge $(u, v) \in E$, the graph G' that we construct will contain one copy of this widget, which we denote by W_{uv}. We denote each vertex in W_{uv} by $[u, v, i]$ or $[v, u, i]$, where $1 \le i \le 6$, so that each widget W_{uv} contains 12 vertices. Widget W_{uv} also contains the 14 edges shown in Figure 34.16(a).

Along with the internal structure of the widget, we enforce the properties we want by limiting the connections between the widget and the remainder of the graph G' that we construct. In particular, only vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$ will have edges incident from outside W_{uv}. Any hamiltonian cycle of G' must traverse the edges of W_{uv} in one of the three ways shown in Figures 34.16(b)–(d). If the cycle enters through vertex $[u, v, 1]$, it must exit through vertex $[u, v, 6]$, and it either visits all 12 of the widget's vertices (Figure 34.16(b)) or the six vertices $[u, v, 1]$ through $[u, v, 6]$ (Figure 34.16(c)). In the latter case, the cycle will have to reenter the widget to visit vertices $[v, u, 1]$ through $[v, u, 6]$. Similarly, if the cycle enters through vertex $[v, u, 1]$, it must exit through vertex $[v, u, 6]$, and it either visits all 12 of the widget's vertices (Figure 34.16(d)) or the six vertices $[v, u, 1]$ through $[v, u, 6]$ (Figure 34.16(c)). No other paths through the widget that visit all 12 vertices are possible. In particular, it is impossible to construct two vertex-disjoint paths, one of which connects $[u, v, 1]$ to $[v, u, 6]$ and the other of which connects $[v, u, 1]$ to $[u, v, 6]$, such that the union of the two paths contains all of the widget's vertices.

(a)



(b)

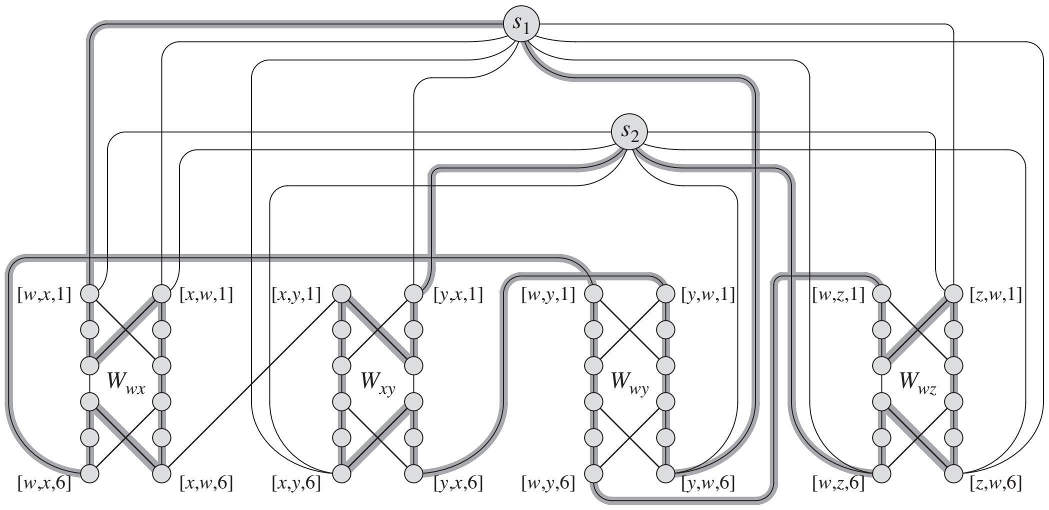


Figure 34.17 Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem. **(a)** An undirected graph G with a vertex cover of size 2, consisting of the lightly shaded vertices w and y. **(b)** The undirected graph G' produced by the reduction, with the hamiltonian path corresponding to the vertex cover shaded. The vertex cover $\{w, y\}$ corresponds to edges $(s_1, [w, x, 1])$ and $(s_2, [y, x, 1])$ appearing in the hamiltonian cycle.

The only other vertices in V' other than those of widgets are ***selector vertices*** $s_1, s_2, \ldots, s_k$. We use edges incident on selector vertices in G' to select the k vertices of the cover in G.

In addition to the edges in widgets, E' contains two other types of edges, which Figure 34.17 shows. First, for each vertex $u \in V$, we add edges to join pairs of widgets in order to form a path containing all widgets corresponding to edges incident on u in G. We arbitrarily order the vertices adjacent to each vertex $u \in V$ as $u^{(1)}, u^{(2)}, \ldots, u^{(\text{degree}(u))}$, where degree$(u)$ is the number of vertices adjacent to u. We create a path in G' through all the widgets corresponding to edges incident on u by adding to E' the edges $\{([u, u^{(i)}, 6], [u, u^{(i+1)}, 1]) : 1 \le i \le \text{degree}(u) - 1\}$. In Figure 34.17, for example, we order the vertices adjacent to w as x, y, z, and so graph G' in part (b) of the figure includes the edges

$([w, x, 6], [w, y, 1])$ and $([w, y, 6], [w, z, 1])$. For each vertex $u \in V$, these edges in G' fill in a path containing all widgets corresponding to edges incident on u in G.

The intuition behind these edges is that if we choose a vertex $u \in V$ in the vertex cover of G, we can construct a path from $[u, u^{(1)}, 1]$ to $[u, u^{(\text{degree}(u))}, 6]$ in G' that "covers" all widgets corresponding to edges incident on u. That is, for each of these widgets, say $W_{u,u^{(i)}}$, the path either includes all 12 vertices (if u is in the vertex cover but $u^{(i)}$ is not) or just the six vertices $[u, u^{(i)}, 1], [u, u^{(i)}, 2], \ldots, [u, u^{(i)}, 6]$ (if both u and $u^{(i)}$ are in the vertex cover).

The final type of edge in E' joins the first vertex $[u, u^{(1)}, 1]$ and the last vertex $[u, u^{(\text{degree}(u))}, 6]$ of each of these paths to each of the selector vertices. That is, we include the edges

$$\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ and } 1 \le j \le k\} \\ \cup \{(s_j, [u, u^{(\text{degree}(u))}, 6]) : u \in V \text{ and } 1 \le j \le k\} .$$

Next, we show that the size of G' is polynomial in the size of G, and hence we can construct G' in time polynomial in the size of G. The vertices of G' are those in the widgets, plus the selector vertices. With 12 vertices per widget, plus $k \le |V|$ selector vertices, we have a total of

$$\begin{aligned} |V'| &= 12\,|E| + k \\ &\le 12\,|E| + |V| \end{aligned}$$

vertices. The edges of G' are those in the widgets, those that go between widgets, and those connecting selector vertices to widgets. Each widget contains 14 edges, totaling $14\,|E|$ in all widgets. For each vertex $u \in V$, graph G' has $\text{degree}(u) - 1$ edges going between widgets, so that summed over all vertices in V,

$$\sum_{u \in V} (\text{degree}(u) - 1) = 2\,|E| - |V|$$

edges go between widgets. Finally, G' has two edges for each pair consisting of a selector vertex and a vertex of V, totaling $2k\,|V|$ such edges. The total number of edges of G' is therefore

$$\begin{aligned} |E'| &= (14\,|E|) + (2\,|E| - |V|) + (2k\,|V|) \\ &= 16\,|E| + (2k - 1)\,|V| \\ &\le 16\,|E| + (2\,|V| - 1)\,|V| \ . \end{aligned}$$

Now we show that the transformation from graph G to G' is a reduction. That is, we must show that G has a vertex cover of size k if and only if G' has a hamiltonian cycle.

Suppose that $G = (V, E)$ has a vertex cover $V^* \subseteq V$ of size k. Let $V^* = \{u_1, u_2, \ldots, u_k\}$. As Figure 34.17 shows, we form a hamiltonian cycle in G' by including the following edges[10] for each vertex $u_j \in V^*$. Include edges $\left\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \le i \le \text{degree}(u_j) - 1\right\}$, which connect all widgets corresponding to edges incident on u_j. We also include the edges within these widgets as Figures 34.16(b)–(d) show, depending on whether the edge is covered by one or two vertices in V^*. The hamiltonian cycle also includes the edges

$$\begin{aligned}&\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \le j \le k\} \\ &\qquad \cup \{(s_{j+1}, [u_j, u_j^{(\text{degree}(u_j))}, 6]) : 1 \le j \le k-1\} \\ &\qquad \cup \{(s_1, [u_k, u_k^{(\text{degree}(u_k))}, 6])\} .\end{aligned}$$

By inspecting Figure 34.17, you can verify that these edges form a cycle. The cycle starts at s_1, visits all widgets corresponding to edges incident on u_1, then visits s_2, visits all widgets corresponding to edges incident on u_2, and so on, until it returns to s_1. The cycle visits each widget either once or twice, depending on whether one or two vertices of V^* cover its corresponding edge. Because V^* is a vertex cover for G, each edge in E is incident on some vertex in V^*, and so the cycle visits each vertex in each widget of G'. Because the cycle also visits every selector vertex, it is hamiltonian.

Conversely, suppose that $G' = (V', E')$ has a hamiltonian cycle $C \subseteq E'$. We claim that the set

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ for some } 1 \le j \le k\} \tag{34.4}$$

is a vertex cover for G. To see why, partition C into maximal paths that start at some selector vertex s_i, traverse an edge $(s_i, [u, u^{(1)}, 1])$ for some $u \in V$, and end at a selector vertex s_j without passing through any other selector vertex. Let us call each such path a "cover path." From how G' is constructed, each cover path must start at some s_i, take the edge $(s_i, [u, u^{(1)}, 1])$ for some vertex $u \in V$, pass through all the widgets corresponding to edges in E incident on u, and then end at some selector vertex s_j. We refer to this cover path as p_u, and by equation (34.4), we put u into V^*. Each widget visited by p_u must be W_{uv} or W_{vu} for some $v \in V$. For each widget visited by p_u, its vertices are visited by either one or two cover paths. If they are visited by one cover path, then edge $(u, v) \in E$ is covered in G by vertex u. If two cover paths visit the widget, then the other cover path must be p_v, which implies that $v \in V^*$, and edge $(u, v) \in E$ is covered by both u and v.

[10]Technically, we define a cycle in terms of vertices rather than edges (see Section B.4). In the interest of clarity, we abuse notation here and define the hamiltonian cycle in terms of edges.

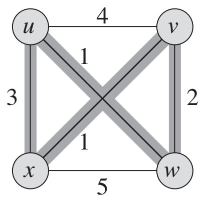


Figure 34.18 An instance of the traveling-salesman problem. Shaded edges represent a minimum-cost tour, with cost 7.

Because each vertex in each widget is visited by some cover path, we see that each edge in E is covered by some vertex in V^*. ■

34.5.4 The traveling-salesman problem

In the ***traveling-salesman problem***, which is closely related to the hamiltonian-cycle problem, a salesman must visit n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a ***tour***, or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. The salesman incurs a nonnegative integer cost $c(i, j)$ to travel from city i to city j, and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. For example, in Figure 34.18, a minimum-cost tour is $\langle u, w, v, x, u \rangle$, with cost 7. The formal language for the corresponding decision problem is

$$\begin{aligned}\text{TSP} = \{\langle G, c, k\rangle : \ & G = (V, E) \text{ is a complete graph,} \\ & c \text{ is a function from } V \times V \to \mathbb{Z}, \\ & k \in \mathbb{Z}, \text{ and} \\ & G \text{ has a traveling-salesman tour with cost at most } k\} \ .\end{aligned}$$

The following theorem shows that a fast algorithm for the traveling-salesman problem is unlikely to exist.

Theorem 34.14
The traveling-salesman problem is NP-complete.

Proof We first show that TSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE $\leq_P$ TSP. Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V$ and $i \neq j\}$, and we define the cost function c by

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \ , \\ 1 & \text{if } (i, j) \notin E \ . \end{cases}$$

(Note that because G is undirected, it has no self-loops, and so $c(v, v) = 1$ for all vertices $v \in V$.) The instance of TSP is then $\langle G', c, 0 \rangle$, which we can easily create in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We conclude that h' is a hamiltonian cycle in graph G. ■

34.5.5 The subset-sum problem

We next consider an arithmetic NP-complete problem. In the ***subset-sum problem***, we are given a finite set S of positive integers and an integer ***target*** $t > 0$. We ask whether there exists a subset $S' \subseteq S$ whose elements sum to t. For example, if $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ and $t = 138457$, then the subset $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ is a solution.

As usual, we define the problem as a language:

$$\text{SUBSET-SUM} = \{\langle S, t \rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \textstyle\sum_{s \in S'} s\} \ .$$

As with any arithmetic problem, it is important to recall that our standard encoding assumes that the input integers are coded in binary. With this assumption in mind, we can show that the subset-sum problem is unlikely to have a fast algorithm.

Theorem 34.15
The subset-sum problem is NP-complete.

Proof To show that SUBSET-SUM is in NP, for an instance $\langle S, t \rangle$ of the problem, we let the subset S' be the certificate. A verification algorithm can check whether $t = \sum_{s \in S'} s$ in polynomial time.

We now show that 3-CNF-SAT $\leq_P$ SUBSET-SUM. Given a 3-CNF formula ϕ over variables $x_1, x_2, \ldots, x_n$ with clauses $C_1, C_2, \ldots, C_k$, each containing exactly three distinct literals, the reduction algorithm constructs an instance $\langle S, t \rangle$ of the subset-sum problem such that ϕ is satisfiable if and only if there exists a subset of S whose sum is exactly t. Without loss of generality, we make two simplifying assumptions about the formula ϕ. First, no clause contains both a variable and its negation, for such a clause is automatically satisfied by any assignment of values to the variables. Second, each variable appears in at least one clause, because it does not matter what value is assigned to a variable that appears in no clauses.

The reduction creates two numbers in set S for each variable x_i and two numbers in S for each clause C_j. We shall create numbers in base 10, where each number contains $n+k$ digits and each digit corresponds to either one variable or one clause. Base 10 (and other bases, as we shall see) has the property we need of preventing carries from lower digits to higher digits.

As Figure 34.19 shows, we construct set S and target t as follows. We label each digit position by either a variable or a clause. The least significant k digits are labeled by the clauses, and the most significant n digits are labeled by variables.

- The target t has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.
- For each variable x_i, set S contains two integers v_i and v'_i. Each of v_i and v'_i has a 1 in the digit labeled by x_i and 0s in the other variable digits. If literal x_i appears in clause C_j, then the digit labeled by C_j in v_i contains a 1. If literal $\neg x_i$ appears in clause C_j, then the digit labeled by C_j in v'_i contains a 1. All other digits labeled by clauses in v_i and v'_i are 0.

  All v_i and v'_i values in set S are unique. Why? For $l \neq i$, no v_l or v'_l values can equal v_i and v'_i in the most significant n digits. Furthermore, by our simplifying assumptions above, no v_i and v'_i can be equal in all k least significant digits. If v_i and v'_i were equal, then x_i and $\neg x_i$ would have to appear in exactly the same set of clauses. But we assume that no clause contains both x_i and $\neg x_i$ and that either x_i or $\neg x_i$ appears in some clause, and so there must be some clause C_j for which v_i and v'_i differ.
- For each clause C_j, set S contains two integers s_j and s'_j. Each of s_j and s'_j has 0s in all digits other than the one labeled by C_j. For s_j, there is a 1 in the C_j digit, and s'_j has a 2 in this digit. These integers are "slack variables," which we use to get each clause-labeled digit position to add to the target value of 4.

  Simple inspection of Figure 34.19 demonstrates that all s_j and s'_j values in S are unique in set S.

Note that the greatest sum of digits in any one digit position is 6, which occurs in the digits labeled by clauses (three 1s from the v_i and v'_i values, plus 1 and 2 from

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Figure 34.19 The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$, and $C_4 = (x_1 \vee x_2 \vee x_3)$. A satisfying assignment of ϕ is $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$. The set S produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. The target t is 1114444. The subset $S' \subseteq S$ is lightly shaded, and it contains v'_1, v'_2, and v_3, corresponding to the satisfying assignment. It also contains slack variables s_1, s'_1, s'_2, s_3, s_4, and s'_4 to achieve the target value of 4 in the digits labeled by C_1 through C_4.

the s_j and s'_j values). Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits.[11]

We can perform the reduction in polynomial time. The set S contains $2n + 2k$ values, each of which has $n + k$ digits, and the time to produce each digit is polynomial in $n + k$. The target t has $n + k$ digits, and the reduction produces each in constant time.

We now show that the 3-CNF formula ϕ is satisfiable if and only if there exists a subset $S' \subseteq S$ whose sum is t. First, suppose that ϕ has a satisfying assignment. For $i = 1, 2, \ldots, n$, if $x_i = 1$ in this assignment, then include v_i in S'. Otherwise, include v'_i. In other words, we include in S' exactly the v_i and v'_i values that cor-

[11]In fact, any base b, where $b \geq 7$, would work. The instance at the beginning of this subsection is the set S and target t in Figure 34.19 interpreted in base 7, with S listed in sorted order.

respond to literals with the value 1 in the satisfying assignment. Having included either v_i or v'_i, but not both, for all i, and having put 0 in the digits labeled by variables in all s_j and s'_j, we see that for each variable-labeled digit, the sum of the values of S' must be 1, which matches those digits of the target t. Because each clause is satisfied, the clause contains some literal with the value 1. Therefore, each digit labeled by a clause has at least one 1 contributed to its sum by a v_i or v'_i value in S'. In fact, 1, 2, or 3 literals may be 1 in each clause, and so each clause-labeled digit has a sum of 1, 2, or 3 from the v_i and v'_i values in S'. In Figure 34.19 for example, literals $\neg x_1$, $\neg x_2$, and x_3 have the value 1 in a satisfying assignment. Each of clauses C_1 and C_4 contains exactly one of these literals, and so together v'_1, v'_2, and v_3 contribute 1 to the sum in the digits for C_1 and C_4. Clause C_2 contains two of these literals, and v'_1, v'_2, and v_3 contribute 2 to the sum in the digit for C_2. Clause C_3 contains all three of these literals, and v'_1, v'_2, and v_3 contribute 3 to the sum in the digit for C_3. We achieve the target of 4 in each digit labeled by clause C_j by including in S' the appropriate nonempty subset of slack variables $\{s_j, s'_j\}$. In Figure 34.19, S' includes s_1, s'_1, s'_2, s_3, s_4, and s'_4. Since we have matched the target in all digits of the sum, and no carries can occur, the values of S' sum to t.

Now, suppose that there is a subset $S' \subseteq S$ that sums to t. The subset S' must include exactly one of v_i and v'_i for each $i = 1, 2, \ldots, n$, for otherwise the digits labeled by variables would not sum to 1. If $v_i \in S'$, we set $x_i = 1$. Otherwise, $v'_i \in S'$, and we set $x_i = 0$. We claim that every clause C_j, for $j = 1, 2, \ldots, k$, is satisfied by this assignment. To prove this claim, note that to achieve a sum of 4 in the digit labeled by C_j, the subset S' must include at least one v_i or v'_i value that has a 1 in the digit labeled by C_j, since the contributions of the slack variables s_j and s'_j together sum to at most 3. If S' includes a v_i that has a 1 in C_j's position, then the literal x_i appears in clause C_j. Since we have set $x_i = 1$ when $v_i \in S'$, clause C_j is satisfied. If S' includes a v'_i that has a 1 in that position, then the literal $\neg x_i$ appears in C_j. Since we have set $x_i = 0$ when $v'_i \in S'$, clause C_j is again satisfied. Thus, all clauses of ϕ are satisfied, which completes the proof. ■

Exercises

34.5-1

The ***subgraph-isomorphism problem*** takes two undirected graphs G_1 and G_2, and it asks whether G_1 is isomorphic to a subgraph of G_2. Show that the subgraph-isomorphism problem is NP-complete.

34.5-2

Given an integer $m \times n$ matrix A and an integer m-vector b, the ***0-1 integer-programming problem*** asks whether there exists an integer n-vector x with elements in the set $\{0,1\}$ such that $Ax \le b$. Prove that 0-1 integer programming is NP-complete. (*Hint:* Reduce from 3-CNF-SAT.)

34.5-3
The ***integer linear-programming problem*** is like the 0-1 integer-programming problem given in Exercise 34.5-2, except that the values of the vector x may be any integers rather than just 0 or 1. Assuming that the 0-1 integer-programming problem is NP-hard, show that the integer linear-programming problem is NP-complete.

34.5-4
Show how to solve the subset-sum problem in polynomial time if the target value t is expressed in unary.

34.5-5
The ***set-partition problem*** takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $\overline{A} = S - A$ such that $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$. Show that the set-partition problem is NP-complete.

34.5-6
Show that the hamiltonian-path problem is NP-complete.

34.5-7
The ***longest-simple-cycle problem*** is the problem of determining a simple cycle (no repeated vertices) of maximum length in a graph. Formulate a related decision problem, and show that the decision problem is NP-complete.

34.5-8
In the ***half 3-CNF satisfiability*** problem, we are given a 3-CNF formula ϕ with n variables and m clauses, where m is even. We wish to determine whether there exists a truth assignment to the variables of ϕ such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete.

Problems

34-1 Independent set
An ***independent set*** of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident on at most one vertex in V'. The ***independent-set problem*** is to find a maximum-size independent set in G.

a. Formulate a related decision problem for the independent-set problem, and prove that it is NP-complete. (*Hint:* Reduce from the clique problem.)

b. Suppose that you are given a "black-box" subroutine to solve the decision problem you defined in part (a). Give an algorithm to find an independent set of maximum size. The running time of your algorithm should be polynomial in $|V|$ and $|E|$, counting queries to the black box as a single step.

Although the independent-set decision problem is NP-complete, certain special cases are polynomial-time solvable.

c. Give an efficient algorithm to solve the independent-set problem when each vertex in G has degree 2. Analyze the running time, and prove that your algorithm works correctly.

d. Give an efficient algorithm to solve the independent-set problem when G is bipartite. Analyze the running time, and prove that your algorithm works correctly. (*Hint:* Use the results of Section 26.3.)

34-2 Bonnie and Clyde

Bonnie and Clyde have just robbed a bank. They have a bag of money and want to divide it up. For each of the following scenarios, either give a polynomial-time algorithm, or prove that the problem is NP-complete. The input in each case is a list of the n items in the bag, along with the value of each.

a. The bag contains n coins, but only 2 different denominations: some coins are worth x dollars, and some are worth y dollars. Bonnie and Clyde wish to divide the money exactly evenly.

b. The bag contains n coins, with an arbitrary number of different denominations, but each denomination is a nonnegative integer power of 2, i.e., the possible denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to divide the money exactly evenly.

c. The bag contains n checks, which are, in an amazing coincidence, made out to "Bonnie or Clyde." They wish to divide the checks so that they each get the exact same amount of money.

d. The bag contains n checks as in part (c), but this time Bonnie and Clyde are willing to accept a split in which the difference is no larger than 100 dollars.

34-3 Graph coloring

Mapmakers try to use as few colors as possible when coloring countries on a map, as long as no two countries that share a border have the same color. We can model this problem with an undirected graph $G = (V, E)$ in which each vertex represents a country and vertices whose respective countries share a border are adjacent. Then, a ***k-coloring*** is a function $c : V \to \{1, 2, \ldots, k\}$ such that $c(u) \neq c(v)$ for every edge $(u, v) \in E$. In other words, the numbers $1, 2, \ldots, k$ represent the k colors, and adjacent vertices must have different colors. The ***graph-coloring problem*** is to determine the minimum number of colors needed to color a given graph.

a. Give an efficient algorithm to determine a 2-coloring of a graph, if one exists.

b. Cast the graph-coloring problem as a decision problem. Show that your decision problem is solvable in polynomial time if and only if the graph-coloring problem is solvable in polynomial time.

c. Let the language 3-COLOR be the set of graphs that can be 3-colored. Show that if 3-COLOR is NP-complete, then your decision problem from part (b) is NP-complete.

To prove that 3-COLOR is NP-complete, we use a reduction from 3-CNF-SAT. Given a formula ϕ of m clauses on n variables $x_1, x_2, \ldots, x_n$, we construct a graph $G = (V, E)$ as follows. The set V consists of a vertex for each variable, a vertex for the negation of each variable, 5 vertices for each clause, and 3 special vertices: TRUE, FALSE, and RED. The edges of the graph are of two types: "literal" edges that are independent of the clauses and "clause" edges that depend on the clauses. The literal edges form a triangle on the special vertices and also form a triangle on x_i, $\neg x_i$, and RED for $i = 1, 2, \ldots, n$.

d. Argue that in any 3-coloring c of a graph containing the literal edges, exactly one of a variable and its negation is colored $c(\text{TRUE})$ and the other is colored $c(\text{FALSE})$. Argue that for any truth assignment for ϕ, there exists a 3-coloring of the graph containing just the literal edges.

The widget shown in Figure 34.20 helps to enforce the condition corresponding to a clause $(x \vee y \vee z)$. Each clause requires a unique copy of the 5 vertices that are heavily shaded in the figure; they connect as shown to the literals of the clause and the special vertex TRUE.

e. Argue that if each of x, y, and z is colored $c(\text{TRUE})$ or $c(\text{FALSE})$, then the widget is 3-colorable if and only if at least one of x, y, or z is colored $c(\text{TRUE})$.

f. Complete the proof that 3-COLOR is NP-complete.

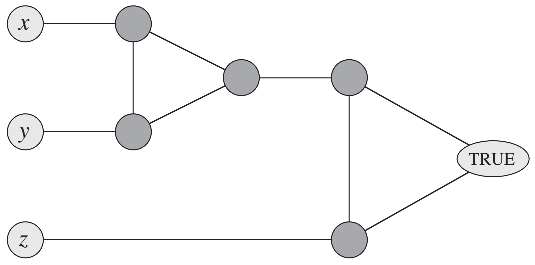


Figure 34.20 The widget corresponding to a clause $(x \vee y \vee z)$, used in Problem 34-3.

34-4 Scheduling with profits and deadlines

Suppose that we have one machine and a set of n tasks $a_1, a_2, \ldots, a_n$, each of which requires time on the machine. Each task a_j requires t_j time units on the machine (its processing time), yields a profit of p_j, and has a deadline d_j. The machine can process only one task at a time, and task a_j must run without interruption for t_j consecutive time units. If we complete task a_j by its deadline d_j, we receive a profit p_j, but if we complete it after its deadline, we receive no profit. As an optimization problem, we are given the processing times, profits, and deadlines for a set of n tasks, and we wish to find a schedule that completes all the tasks and returns the greatest amount of profit. The processing times, profits, and deadlines are all nonnegative numbers.

a. State this problem as a decision problem.

b. Show that the decision problem is NP-complete.

c. Give a polynomial-time algorithm for the decision problem, assuming that all processing times are integers from 1 to n. (*Hint:* Use dynamic programming.)

d. Give a polynomial-time algorithm for the optimization problem, assuming that all processing times are integers from 1 to n.

Chapter notes

The book by Garey and Johnson [129] provides a wonderful guide to NP-completeness, discussing the theory at length and providing a catalogue of many problems that were known to be NP-complete in 1979. The proof of Theorem 34.13 is adapted from their book, and the list of NP-complete problem domains at the beginning of Section 34.5 is drawn from their table of contents. Johnson wrote a series of 23 columns in the *Journal of Algorithms* between 1981 and 1992 reporting new developments in NP-completeness. Hopcroft, Motwani, and Ullman [177], Lewis and Papadimitriou [236], Papadimitriou [270], and Sipser [317] have good treatments of NP-completeness in the context of complexity theory. NP-completeness and several reductions also appear in books by Aho, Hopcroft, and Ullman [5]; Dasgupta, Papadimitriou, and Vazirani [82]; Johnsonbaugh and Schaefer [193]; and Kleinberg and Tardos [208].

The class P was introduced in 1964 by Cobham [72] and, independently, in 1965 by Edmonds [100], who also introduced the class NP and conjectured that P $\neq$ NP. The notion of NP-completeness was proposed in 1971 by Cook [75], who gave the first NP-completeness proofs for formula satisfiability and 3-CNF satisfiability. Levin [234] independently discovered the notion, giving an NP-completeness proof for a tiling problem. Karp [199] introduced the methodology of reductions in 1972 and demonstrated the rich variety of NP-complete problems. Karp's paper included the original NP-completeness proofs of the clique, vertex-cover, and hamiltonian-cycle problems. Since then, thousands of problems have been proven to be NP-complete by many researchers. In a talk at a meeting celebrating Karp's 60th birthday in 1995, Papadimitriou remarked, "about 6000 papers each year have the term 'NP-complete' on their title, abstract, or list of keywords. This is more than each of the terms 'compiler,' 'database,' 'expert,' 'neural network,' or 'operating system.' "

Recent work in complexity theory has shed light on the complexity of computing approximate solutions. This work gives a new definition of NP using "probabilistically checkable proofs." This new definition implies that for problems such as clique, vertex cover, the traveling-salesman problem with the triangle inequality, and many others, computing good approximate solutions is NP-hard and hence no easier than computing optimal solutions. An introduction to this area can be found in Arora's thesis [20]; a chapter by Arora and Lund in Hochbaum [172]; a survey article by Arora [21]; a book edited by Mayr, Prömel, and Steger [246]; and a survey article by Johnson [191].

Approximation Algorithms

35 Approximation Algorithms

Many problems of practical significance are NP-complete, yet they are too important to abandon merely because we don't know how to find an optimal solution in polynomial time. Even if a problem is NP-complete, there may be hope. We have at least three ways to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that we can solve in polynomial time. Third, we might come up with approaches to find *near-optimal* solutions in polynomial time (either in the worst case or the expected case). In practice, near-optimality is often good enough. We call an algorithm that returns near-optimal solutions an ***approximation algorithm***. This chapter presents polynomial-time approximation algorithms for several NP-complete problems.

Performance ratios for approximation algorithms

Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution. Depending on the problem, we may define an optimal solution as one with maximum possible cost or one with minimum possible cost; that is, the problem may be either a maximization or a minimization problem.

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) . \qquad (35.1)$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a ***$\rho(n)$-approximation algorithm***. The definitions of the approximation ratio and of a $\rho(n)$-approximation algorithm apply to both minimization and maximization problems. For a maximization problem, $0 < C \le C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. Because we assume that all solutions have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* \leq 1$ implies $C^*/C \geq 1$. Therefore, a 1-approximation algorithm[1] produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

For many problems, we have polynomial-time approximation algorithms with small constant approximation ratios, although for other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size n. An example of such a problem is the set-cover problem presented in Section 35.3.

Some NP-complete problems allow polynomial-time approximation algorithms that can achieve increasingly better approximation ratios by using more and more computation time. That is, we can trade computation time for the quality of the approximation. An example is the subset-sum problem studied in Section 35.5. This situation is important enough to deserve a name of its own.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

The running time of a polynomial-time approximation scheme can increase very rapidly as ϵ decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\epsilon})$. Ideally, if ϵ decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor (though not necessarily the same constant factor by which ϵ decreased).

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** if it is an approximation scheme and its running time is polynomial in both $1/\epsilon$ and the size n of the input instance. For example, the scheme might have a running time of $O((1/\epsilon)^2 n^3)$. With such a scheme, any constant-factor decrease in ϵ comes with a corresponding constant-factor increase in the running time.

[1]When the approximation ratio is independent of n, we use the terms "approximation ratio of ρ" and "ρ-approximation algorithm," indicating no dependence on n.

Chapter outline

The first four sections of this chapter present some examples of polynomial-time approximation algorithms for NP-complete problems, and the fifth section presents a fully polynomial-time approximation scheme. Section 35.1 begins with a study of the vertex-cover problem, an NP-complete minimization problem that has an approximation algorithm with an approximation ratio of 2. Section 35.2 presents an approximation algorithm with an approximation ratio of 2 for the case of the traveling-salesman problem in which the cost function satisfies the triangle inequality. It also shows that without the triangle inequality, for any constant $\rho \geq 1$, a ρ-approximation algorithm cannot exist unless $\mathrm{P} = \mathrm{NP}$. In Section 35.3, we show how to use a greedy method as an effective approximation algorithm for the set-covering problem, obtaining a covering whose cost is at worst a logarithmic factor larger than the optimal cost. Section 35.4 presents two more approximation algorithms. First we study the optimization version of 3-CNF satisfiability and give a simple randomized algorithm that produces a solution with an expected approximation ratio of $8/7$. Then we examine a weighted variant of the vertex-cover problem and show how to use linear programming to develop a 2-approximation algorithm. Finally, Section 35.5 presents a fully polynomial-time approximation scheme for the subset-sum problem.

35.1 The vertex-cover problem

Section 34.5.2 defined the vertex-cover problem and proved it NP-complete. Recall that a ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

Even though we don't know how to find an optimal vertex cover in a graph G in polynomial time, we can efficiently find a vertex cover that is near-optimal. The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

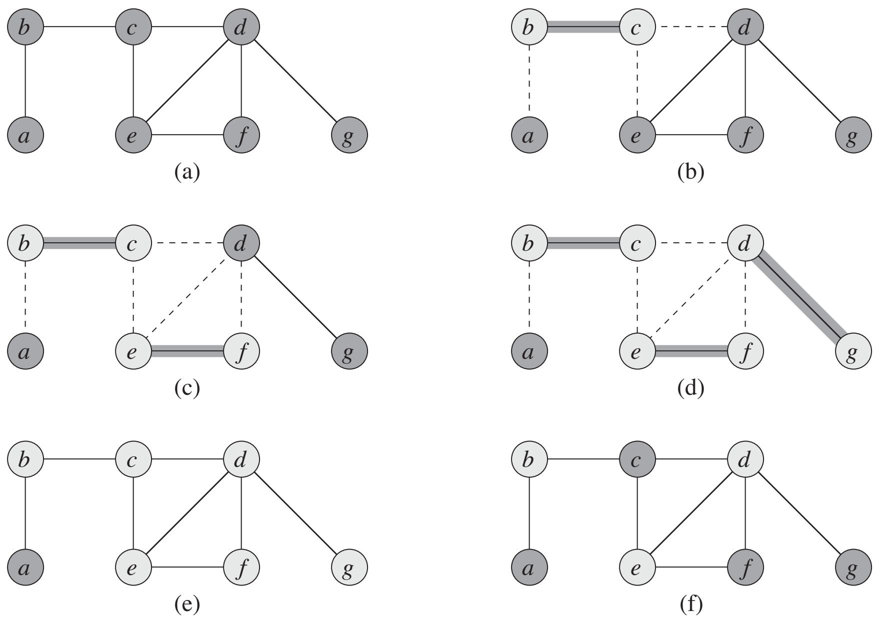


Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

```
APPROX-VERTEX-COVER($G$)
1  $C = \emptyset$
2  $E' = G.E$
3  while $E' \neq \emptyset$
4      let $(u, v)$ be an arbitrary edge of $E'$
5      $C = C \cup \{u, v\}$
6      remove from $E'$ every edge incident on either $u$ or $v$
7  return $C$
```

Figure 35.1 illustrates how APPROX-VERTEX-COVER operates on an example graph. The variable C contains the vertex cover being constructed. Line 1 initializes C to the empty set. Line 2 sets E' to be a copy of the edge set $G.E$ of the graph. The loop of lines 3–6 repeatedly picks an edge (u, v) from E', adds its endpoints u and v to C, and deletes all edges in E' that are covered by either u or v. Finally, line 7 returns the vertex cover C. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E'.

Theorem 35.1
APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof We have already shown that APPROX-VERTEX-COVER runs in polynomial time.

The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $G.E$ has been covered by some vertex in C.

To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that line 4 of APPROX-VERTEX-COVER picked. In order to cover the edges in A, any vertex cover—in particular, an optimal cover C^*—must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6. Thus, no two edges in A are covered by the same vertex from C^*, and we have the lower bound

$$|C^*| \geq |A| \tag{35.2}$$

on the size of an optimal vertex cover. Each execution of line 4 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2\,|A| \ . \tag{35.3}$$

Combining equations (35.2) and (35.3), we obtain

$$\begin{aligned} |C| &= 2\,|A| \\ &\leq 2\,|C^*| \ , \end{aligned}$$

thereby proving the theorem. ■

Let us reflect on this proof. At first, you might wonder how we can possibly prove that the size of the vertex cover returned by APPROX-VERTEX-COVER is at most twice the size of an optimal vertex cover, when we do not even know the size of an optimal vertex cover. Instead of requiring that we know the exact size of an optimal vertex cover, we rely on a lower bound on the size. As Exercise 35.1-2 asks you to show, the set A of edges that line 4 of APPROX-VERTEX-COVER selects is actually a maximal matching in the graph G. (A ***maximal matching*** is a matching that is not a proper subset of any other matching.) The size of a maximal matching is, as we argued in the proof of Theorem 35.1, a lower bound on the size of an optimal vertex cover. The algorithm returns a vertex cover whose size is at most twice the size of the maximal matching A. By relating the size of the solution returned to the lower bound, we obtain our approximation ratio. We will use this methodology in later sections as well.

Exercises

35.1-1
Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.

35.1-2
Prove that the set of edges picked in line 4 of APPROX-VERTEX-COVER forms a maximal matching in the graph G.

35.1-3 ⋆
Professor Bündchen proposes the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that the professor's heuristic does not have an approximation ratio of 2. (*Hint:* Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.)

35.1-4
Give an efficient greedy algorithm that finds an optimal vertex cover for a tree in linear time.

35.1-5
From the proof of Theorem 34.12, we know that the vertex-cover problem and the NP-complete clique problem are complementary in the sense that an optimal vertex cover is the complement of a maximum-size clique in the complement graph. Does this relationship imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for the clique problem? Justify your answer.

35.2 The traveling-salesman problem

In the traveling-salesman problem introduced in Section 34.5.4, we are given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, and we must find a hamiltonian cycle (a tour) of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$:

$$c(A) = \sum_{(u,v)\in A} c(u,v) .$$

In many practical situations, the least costly way to go from a place u to a place w is to go directly, with no intermediate steps. Put another way, cutting out an intermediate stop never increases the cost. We formalize this notion by saying that the cost function c satisfies the ***triangle inequality*** if, for all vertices $u, v, w \in V$,

$$c(u,w) \leq c(u,v) + c(v,w) .$$

The triangle inequality seems as though it should naturally hold, and it is automatically satisfied in several applications. For example, if the vertices of the graph are points in the plane and the cost of traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied. Furthermore, many cost functions other than euclidean distance satisfy the triangle inequality.

As Exercise 35.2-2 shows, the traveling-salesman problem is NP-complete even if we require that the cost function satisfy the triangle inequality. Thus, we should not expect to find a polynomial-time algorithm for solving this problem exactly. Instead, we look for good approximation algorithms.

In Section 35.2.1, we examine a 2-approximation algorithm for the traveling-salesman problem with the triangle inequality. In Section 35.2.2, we show that without the triangle inequality, a polynomial-time approximation algorithm with a constant approximation ratio does not exist unless P = NP.

35.2.1 The traveling-salesman problem with the triangle inequality

Applying the methodology of the previous section, we shall first compute a structure—a minimum spanning tree—whose weight gives a lower bound on the length of an optimal traveling-salesman tour. We shall then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality. The following algorithm implements this approach, calling the minimum-spanning-tree algorithm MST-PRIM from Section 23.2 as a subroutine. The parameter G is a complete undirected graph, and the cost function c satisfies the triangle inequality.

```
APPROX-TSP-TOUR($G, c$)
1  select a vertex $r \in G.V$ to be a "root" vertex
2  compute a minimum spanning tree $T$ for $G$ from root $r$
       using MST-PRIM($G, c, r$)
3  let $H$ be a list of vertices, ordered according to when they are first visited
       in a preorder tree walk of $T$
4  return the hamiltonian cycle $H$
```

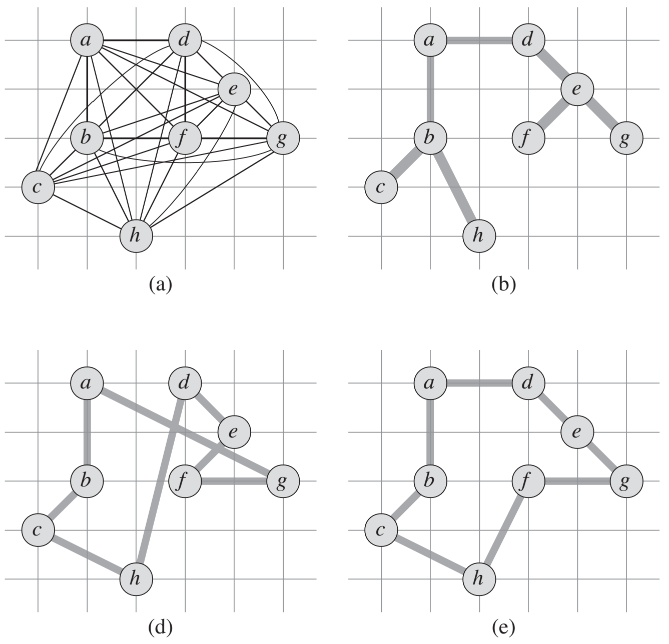


Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the original complete graph. Its total cost is approximately 14.715.

Recall from Section 12.1 that a preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children.

Figure 35.2 illustrates the operation of APPROX-TSP-TOUR. Part (a) of the figure shows a complete undirected graph, and part (b) shows the minimum spanning tree T grown from root vertex a by MST-PRIM. Part (c) shows how a preorder walk of T visits the vertices, and part (d) displays the corresponding tour, which is the tour returned by APPROX-TSP-TOUR. Part (e) displays an optimal tour, which is about 23% shorter.

By Exercise 23.2-2, even with a simple implementation of MST-PRIM, the running time of APPROX-TSP-TOUR is $\Theta(V^2)$. We now show that if the cost function for an instance of the traveling-salesman problem satisfies the triangle inequality, then APPROX-TSP-TOUR returns a tour whose cost is not more than twice the cost of an optimal tour.

Theorem 35.2
APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality.

Proof We have already seen that APPROX-TSP-TOUR runs in polynomial time.

Let H^* denote an optimal tour for the given set of vertices. We obtain a spanning tree by deleting any edge from a tour, and each edge cost is nonnegative. Therefore, the weight of the minimum spanning tree T computed in line 2 of APPROX-TSP-TOUR provides a lower bound on the cost of an optimal tour:

$$c(T) \le c(H^*) . \tag{35.4}$$

A ***full walk*** of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let us call this full walk W. The full walk of our example gives the order

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

Since the full walk traverses every edge of T exactly twice, we have (extending our definition of the cost c in the natural manner to handle multisets of edges)

$$c(W) = 2c(T) . \tag{35.5}$$

Inequality (35.4) and equation (35.5) imply that

$$c(W) \le 2c(H^*) , \tag{35.6}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour.

Unfortunately, the full walk W is generally not a tour, since it visits some vertices more than once. By the triangle inequality, however, we can delete a visit to any vertex from W and the cost does not increase. (If we delete a vertex v from W between visits to u and w, the resulting ordering specifies going directly from u to w.) By repeatedly applying this operation, we can remove from W all but the first visit to each vertex. In our example, this leaves the ordering

$$a, b, c, h, d, e, f, g .$$

This ordering is the same as that obtained by a preorder walk of the tree T. Let H be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since every vertex is visited exactly once, and in fact it is the cycle computed by APPROX-TSP-TOUR. Since H is obtained by deleting vertices from the full walk W, we have

$$c(H) \leq c(W) . \tag{35.7}$$

Combining inequalities (35.6) and (35.7) gives $c(H) \leq 2c(H^*)$, which completes the proof. ∎

In spite of the nice approximation ratio provided by Theorem 35.2, APPROX-TSP-TOUR is usually not the best practical choice for this problem. There are other approximation algorithms that typically perform much better in practice. (See the references at the end of this chapter.)

35.2.2 The general traveling-salesman problem

If we drop the assumption that the cost function c satisfies the triangle inequality, then we cannot find good approximate tours in polynomial time unless P = NP.

Theorem 35.3
If P $\neq$ NP, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesman problem.

Proof The proof is by contradiction. Suppose to the contrary that for some number $\rho \geq 1$, there is a polynomial-time approximation algorithm A with approximation ratio ρ. Without loss of generality, we assume that ρ is an integer, by rounding it up if necessary. We shall then show how to use A to solve instances of the hamiltonian-cycle problem (defined in Section 34.2) in polynomial time. Since Theorem 34.13 tells us that the hamiltonian-cycle problem is NP-complete, Theorem 34.4 implies that if we can solve it in polynomial time, then P = NP.

Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem. We wish to determine efficiently whether G contains a hamiltonian cycle by making use of the hypothesized approximation algorithm A. We turn G into an instance of the traveling-salesman problem as follows. Let $G' = (V, E')$ be the complete graph on V; that is,

$$E' = \{(u, v) : u, v \in V \text{ and } u \neq v\} .$$

Assign an integer cost to each edge in E' as follows:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E , \\ \rho\,|V| + 1 & \text{otherwise} . \end{cases}$$

We can create representations of G' and c from a representation of G in time polynomial in $|V|$ and $|E|$.

Now, consider the traveling-salesman problem (G', c). If the original graph G has a hamiltonian cycle H, then the cost function c assigns to each edge of H a cost of 1, and so (G', c) contains a tour of cost $|V|$. On the other hand, if G does not contain a hamiltonian cycle, then any tour of G' must use some edge not in E. But any tour that uses an edge not in E has a cost of at least

$$\begin{aligned}(\rho\,|V| + 1) + (|V| - 1) &= \rho\,|V| + |V| \\ &> \rho\,|V| \;.\end{aligned}$$

Because edges not in G are so costly, there is a gap of at least $\rho\,|V|$ between the cost of a tour that is a hamiltonian cycle in G (cost $|V|$) and the cost of any other tour (cost at least $\rho\,|V| + |V|$). Therefore, the cost of a tour that is not a hamiltonian cycle in G is at least a factor of $\rho + 1$ greater than the cost of a tour that is a hamiltonian cycle in G.

Now, suppose that we apply the approximation algorithm A to the traveling-salesman problem (G', c). Because A is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour, if G contains a hamiltonian cycle, then A must return it. If G has no hamiltonian cycle, then A returns a tour of cost more than $\rho\,|V|$. Therefore, we can use A to solve the hamiltonian-cycle problem in polynomial time. ■

The proof of Theorem 35.3 serves as an example of a general technique for proving that we cannot approximate a problem very well. Suppose that given an NP-hard problem X, we can produce in polynomial time a minimization problem Y such that "yes" instances of X correspond to instances of Y with value at most k (for some k), but that "no" instances of X correspond to instances of Y with value greater than ρk. Then, we have shown that, unless P $=$ NP, there is no polynomial-time ρ-approximation algorithm for problem Y.

Exercises

35.2-1

Suppose that a complete undirected graph $G = (V, E)$ with at least 3 vertices has a cost function c that satisfies the triangle inequality. Prove that $c(u, v) \geq 0$ for all $u, v \in V$.

35.2-2

Show how in polynomial time we can transform one instance of the traveling-salesman problem into another instance whose cost function satisfies the triangle inequality. The two instances must have the same set of optimal tours. Explain why such a polynomial-time transformation does not contradict Theorem 35.3, assuming that P $\neq$ NP.

35.2-3

Consider the following ***closest-point heuristic*** for building an approximate traveling-salesman tour whose cost function satisfies the triangle inequality. Begin with a trivial cycle consisting of a single arbitrarily chosen vertex. At each step, identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v. Extend the cycle to include u by inserting u just after v. Repeat until all vertices are on the cycle. Prove that this heuristic returns a tour whose total cost is not more than twice the cost of an optimal tour.

35.2-4

In the ***bottleneck traveling-salesman problem***, we wish to find the hamiltonian cycle that minimizes the cost of the most costly edge in the cycle. Assuming that the cost function satisfies the triangle inequality, show that there exists a polynomial-time approximation algorithm with approximation ratio 3 for this problem. (*Hint:* Show recursively that we can visit all the nodes in a bottleneck spanning tree, as discussed in Problem 23-3, exactly once by taking a full walk of the tree and skipping nodes, but without skipping more than two consecutive intermediate nodes. Show that the costliest edge in a bottleneck spanning tree has a cost that is at most the cost of the costliest edge in a bottleneck hamiltonian cycle.)

35.2-5

Suppose that the vertices for an instance of the traveling-salesman problem are points in the plane and that the cost $c(u, v)$ is the euclidean distance between points u and v. Show that an optimal tour never crosses itself.

35.3 The set-covering problem

The set-covering problem is an optimization problem that models many problems that require resources to be allocated. Its corresponding decision problem generalizes the NP-complete vertex-cover problem and is therefore also NP-hard. The approximation algorithm developed to handle the vertex-cover problem doesn't apply here, however, and so we need to try other approaches. We shall examine a simple greedy heuristic with a logarithmic approximation ratio. That is, as the size of the instance gets larger, the size of the approximate solution may grow, relative to the size of an optimal solution. Because the logarithm function grows rather slowly, however, this approximation algorithm may nonetheless give useful results.

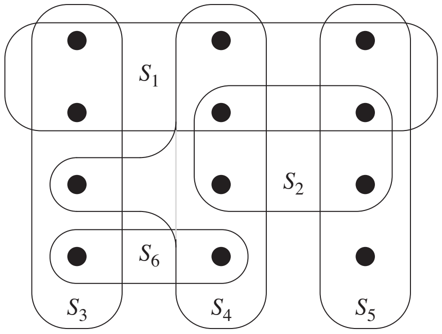


Figure 35.3 An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 black points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S_1, S_4, S_5, and S_3 or the sets S_1, S_4, S_5, and S_6, in order.

An instance $(X, \mathcal{F})$ of the ***set-covering problem*** consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

We say that a subset $S \in \mathcal{F}$ ***covers*** its elements. The problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S . \tag{35.8}$$

We say that any $\mathcal{C}$ satisfying equation (35.8) ***covers*** X. Figure 35.3 illustrates the set-covering problem. The size of $\mathcal{C}$ is the number of sets it contains, rather than the number of individual elements in these sets, since every subset $\mathcal{C}$ that covers X must contain all $|X|$ individual elements. In Figure 35.3, the minimum set cover has size 3.

The set-covering problem abstracts many commonly arising combinatorial problems. As a simple example, suppose that X represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem. We wish to form a committee, containing as few people as possible, such that for every requisite skill in X, at least one member of the committee has that skill. In the decision version of the set-covering problem, we ask whether a covering exists with size at most k, where k is an additional parameter specified in the problem instance. The decision version of the problem is NP-complete, as Exercise 35.3-2 asks you to show.

A greedy approximation algorithm

The greedy method works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered.

```
GREEDY-SET-COVER($X, \mathcal{F}$)
1  $U = X$
2  $\mathcal{C} = \emptyset$
3  while $U \neq \emptyset$
4      select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5      $U = U - S$
6      $\mathcal{C} = \mathcal{C} \cup \{S\}$
7  return $\mathcal{C}$
```

In the example of Figure 35.3, GREEDY-SET-COVER adds to $\mathcal{C}$, in order, the sets S_1, S_4, and S_5, followed by either S_3 or S_6.

The algorithm works as follows. The set U contains, at each stage, the set of remaining uncovered elements. The set $\mathcal{C}$ contains the cover being constructed. Line 4 is the greedy decision-making step, choosing a subset S that covers as many uncovered elements as possible (breaking ties arbitrarily). After S is selected, line 5 removes its elements from U, and line 6 places S into $\mathcal{C}$. When the algorithm terminates, the set $\mathcal{C}$ contains a subfamily of $\mathcal{F}$ that covers X.

We can easily implement GREEDY-SET-COVER to run in time polynomial in $|X|$ and $|\mathcal{F}|$. Since the number of iterations of the loop on lines 3–6 is bounded from above by $\min(|X|, |\mathcal{F}|)$, and we can implement the loop body to run in time $O(|X|\,|\mathcal{F}|)$, a simple implementation runs in time $O(|X|\,|\mathcal{F}|\min(|X|, |\mathcal{F}|))$. Exercise 35.3-3 asks for a linear-time algorithm.

Analysis

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the dth harmonic number $H_d = \sum_{i=1}^{d} 1/i$ (see Section A.1) by $H(d)$. As a boundary condition, we define $H(0) = 0$.

Theorem 35.4
GREEDY-SET-COVER is a polynomial-time $\rho(n)$-approximation algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) .$$

Proof We have already shown that GREEDY-SET-COVER runs in polynomial time.

To show that GREEDY-SET-COVER is a $\rho(n)$-approximation algorithm, we assign a cost of 1 to each set selected by the algorithm, distribute this cost over the elements covered for the first time, and then use these costs to derive the desired relationship between the size of an optimal set cover $\mathcal{C}^*$ and the size of the set cover $\mathcal{C}$ returned by the algorithm. Let S_i denote the ith subset selected by GREEDY-SET-COVER; the algorithm incurs a cost of 1 when it adds S_i to $\mathcal{C}$. We spread this cost of selecting S_i evenly among the elements covered for the first time by S_i. Let c_x denote the cost allocated to element x, for each $x \in X$. Each element is assigned a cost only once, when it is covered for the first time. If x is covered for the first time by S_i, then

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Each step of the algorithm assigns 1 unit of cost, and so

$$|\mathcal{C}| = \sum_{x \in X} c_x . \tag{35.9}$$

Each element $x \in X$ is in at least one set in the optimal cover $\mathcal{C}^*$, and so we have

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x . \tag{35.10}$$

Combining equation (35.9) and inequality (35.10), we have that

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x . \tag{35.11}$$

The remainder of the proof rests on the following key inequality, which we shall prove shortly. For any set S belonging to the family $\mathcal{F}$,

$$\sum_{x \in S} c_x \leq H(|S|) . \tag{35.12}$$

From inequalities (35.11) and (35.12), it follows that

$$\begin{aligned} |\mathcal{C}| &\leq \sum_{S \in \mathcal{C}^*} H(|S|) \\ &\leq |\mathcal{C}^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) , \end{aligned}$$

thus proving the theorem.

All that remains is to prove inequality (35.12). Consider any set $S \in \mathcal{F}$ and any $i = 1, 2, \ldots, |\mathcal{C}|$, and let

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$$

be the number of elements in S that remain uncovered after the algorithm has selected sets $S_1, S_2, \ldots, S_i$. We define $u_0 = |S|$ to be the number of elements of S, which are all initially uncovered. Let k be the least index such that $u_k = 0$, so that every element in S is covered by at least one of the sets $S_1, S_2, \ldots, S_k$ and some element in S is uncovered by $S_1 \cup S_2 \cup \cdots \cup S_{k-1}$. Then, $u_{i-1} \geq u_i$, and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i, for $i = 1, 2, \ldots, k$. Thus,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Observe that

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| &\geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \\ &= u_{i-1} , \end{aligned}$$

because the greedy choice of S_i guarantees that S cannot cover more new elements than S_i does (otherwise, the algorithm would have chosen S instead of S_i). Consequently, we obtain

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} .$$

We now bound this quantity as follows:

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\ &= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\ &\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} && \text{(because } j \leq u_{i-1}\text{)} \\ &= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) && \text{(because the sum telescopes)} \\ &= H(u_0) - H(0) \\ &= H(u_0) && \text{(because } H(0) = 0\text{)} \\ &= H(|S|) , \end{aligned}$$

which completes the proof of inequality (35.12). ■

Corollary 35.5
GREEDY-SET-COVER is a polynomial-time $(\ln |X| + 1)$-approximation algorithm.

Proof Use inequality (A.14) and Theorem 35.4. ■

In some applications, $\max\{|S| : S \in \mathcal{F}\}$ is a small constant, and so the solution returned by GREEDY-SET-COVER is at most a small constant times larger than optimal. One such application occurs when this heuristic finds an approximate vertex cover for a graph whose vertices have degree at most 3. In this case, the solution found by GREEDY-SET-COVER is not more than $H(3) = 11/6$ times as large as an optimal solution, a performance guarantee that is slightly better than that of APPROX-VERTEX-COVER.

Exercises

35.3-1
Consider each of the following words as a set of letters: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}. Show which set cover GREEDY-SET-COVER produces when we break ties in favor of the word that appears first in the dictionary.

35.3-2
Show that the decision version of the set-covering problem is NP-complete by reducing it from the vertex-cover problem.

35.3-3
Show how to implement GREEDY-SET-COVER in such a way that it runs in time $O\left(\sum_{S \in \mathcal{F}} |S|\right)$.

35.3-4
Show that the following weaker form of Theorem 35.4 is trivially true:

$$|\mathcal{C}| \le |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\} \ .$$

35.3-5
GREEDY-SET-COVER can return a number of different solutions, depending on how we break ties in line 4. Give a procedure BAD-SET-COVER-INSTANCE(n) that returns an n-element instance of the set-covering problem for which, depending on how we break ties in line 4, GREEDY-SET-COVER can return a number of different solutions that is exponential in n.

35.4 Randomization and linear programming

In this section, we study two useful techniques for designing approximation algorithms: randomization and linear programming. We shall give a simple randomized algorithm for an optimization version of 3-CNF satisfiability, and then we shall use linear programming to help design an approximation algorithm for a weighted version of the vertex-cover problem. This section only scratches the surface of these two powerful techniques. The chapter notes give references for further study of these areas.

A randomized approximation algorithm for MAX-3-CNF satisfiability

Just as some randomized algorithms compute exact solutions, some randomized algorithms compute approximate solutions. We say that a randomized algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the *expected* cost C of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) . \tag{35.13}$$

We call a randomized algorithm that achieves an approximation ratio of $\rho(n)$ a ***randomized $\rho(n)$-approximation algorithm.*** In other words, a randomized approximation algorithm is like a deterministic approximation algorithm, except that the approximation ratio is for an expected cost.

A particular instance of 3-CNF satisfiability, as defined in Section 34.4, may or may not be satisfiable. In order to be satisfiable, there must exist an assignment of the variables so that every clause evaluates to 1. If an instance is not satisfiable, we may want to compute how "close" to satisfiable it is, that is, we may wish to find an assignment of the variables that satisfies as many clauses as possible. We call the resulting maximization problem ***MAX-3-CNF satisfiability***. The input to MAX-3-CNF satisfiability is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximizes the number of clauses evaluating to 1. We now show that randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ yields a randomized $8/7$-approximation algorithm. According to the definition of 3-CNF satisfiability from Section 34.4, we require each clause to consist of exactly three distinct literals. We further assume that no clause contains both a variable and its negation. (Exercise 35.4-1 asks you to remove this last assumption.)

Theorem 35.6
Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ is a randomized $8/7$-approximation algorithm.

Proof Suppose that we have independently set each variable to 1 with probability $1/2$ and to 0 with probability $1/2$. For $i = 1, 2, \ldots, m$, we define the indicator random variable

$$Y_i = \mathrm{I}\{\text{clause } i \text{ is satisfied}\} ,$$

so that $Y_i = 1$ as long as we have set at least one of the literals in the ith clause to 1. Since no literal appears more than once in the same clause, and since we have assumed that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so $\Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8$. Thus, we have $\Pr\{\text{clause } i \text{ is satisfied}\} = 1 - 1/8 = 7/8$, and by Lemma 5.1, we have $\mathrm{E}[Y_i] = 7/8$. Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Y_m$. Then, we have

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\
&= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{m} 7/8 \\
&= 7m/8 .
\end{aligned}$$

Clearly, m is an upper bound on the number of satisfied clauses, and hence the approximation ratio is at most $m/(7m/8) = 8/7$. ■

Approximating weighted vertex cover using linear programming

In the ***minimum-weight vertex-cover problem***, we are given an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. For any vertex cover $V' \subseteq V$, we define the weight of the vertex cover $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight.

We cannot apply the algorithm used for unweighted vertex cover, nor can we use a random solution; both methods may return solutions that are far from optimal. We shall, however, compute a lower bound on the weight of the minimum-weight vertex cover, by using a linear program. We shall then "round" this solution and use it to obtain a vertex cover.

Suppose that we associate a variable $x(v)$ with each vertex $v \in V$, and let us require that $x(v)$ equals either 0 or 1 for each $v \in V$. We put v into the vertex cover if and only if $x(v) = 1$. Then, we can write the constraint that for any edge (u, v), at least one of u and v must be in the vertex cover as $x(u) + x(v) \geq 1$. This view gives rise to the following ***0-1 integer program*** for finding a minimum-weight vertex cover:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.14}$$

subject to

$$x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \tag{35.15}$$
$$x(v) \in \{0, 1\} \quad \text{for each } v \in V\,. \tag{35.16}$$

In the special case in which all the weights $w(v)$ are equal to 1, this formulation is the optimization version of the NP-hard vertex-cover problem. Suppose, however, that we remove the constraint that $x(v) \in \{0, 1\}$ and replace it by $0 \leq x(v) \leq 1$. We then obtain the following linear program, which is known as the ***linear-programming relaxation***:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.17}$$

subject to

$$x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \tag{35.18}$$
$$x(v) \leq 1 \quad \text{for each } v \in V \tag{35.19}$$
$$x(v) \geq 0 \quad \text{for each } v \in V\,. \tag{35.20}$$

Any feasible solution to the 0-1 integer program in lines (35.14)–(35.16) is also a feasible solution to the linear program in lines (35.17)–(35.20). Therefore, the value of an optimal solution to the linear program gives a lower bound on the value of an optimal solution to the 0-1 integer program, and hence a lower bound on the optimal weight in the minimum-weight vertex-cover problem.

The following procedure uses the solution to the linear-programming relaxation to construct an approximate solution to the minimum-weight vertex-cover problem:

```
APPROX-MIN-WEIGHT-VC$(G, w)$
1  $C = \emptyset$
2  compute $\bar{x}$, an optimal solution to the linear program in lines (35.17)–(35.20)
3  for each $v \in V$
4      if $\bar{x}(v) \geq 1/2$
5          $C = C \cup \{v\}$
6  return $C$
```

The APPROX-MIN-WEIGHT-VC procedure works as follows. Line 1 initializes the vertex cover to be empty. Line 2 formulates the linear program in lines (35.17)–(35.20) and then solves this linear program. An optimal solution gives each vertex v an associated value $\bar{x}(v)$, where $0 \leq \bar{x}(v) \leq 1$. We use this value to guide the choice of which vertices to add to the vertex cover C in lines 3–5. If $\bar{x}(v) \geq 1/2$, we add v to C; otherwise we do not. In effect, we are "rounding" each fractional variable in the solution to the linear program to 0 or 1 in order to obtain a solution to the 0-1 integer program in lines (35.14)–(35.16). Finally, line 6 returns the vertex cover C.

Theorem 35.7
Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the linear program in line 2, and because the **for** loop of lines 3–5 runs in polynomial time, APPROX-MIN-WEIGHT-VC is a polynomial-time algorithm.

Now we show that APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. Let C^* be an optimal solution to the minimum-weight vertex-cover problem, and let z^* be the value of an optimal solution to the linear program in lines (35.17)–(35.20). Since an optimal vertex cover is a feasible solution to the linear program, z^* must be a lower bound on $w(C^*)$, that is,

$$z^* \leq w(C^*) . \tag{35.21}$$

Next, we claim that by rounding the fractional values of the variables $\bar{x}(v)$, we produce a set C that is a vertex cover and satisfies $w(C) \leq 2z^*$. To see that C is a vertex cover, consider any edge $(u, v) \in E$. By constraint (35.18), we know that $x(u) + x(v) \geq 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$. Therefore, at least one of u and v is included in the vertex cover, and so every edge is covered.

Now, we consider the weight of the cover. We have

$$
\begin{aligned}
z^* &= \sum_{v \in V} w(v)\, \bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\, \bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2} \sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C) . \qquad (35.22)
\end{aligned}
$$

Combining inequalities (35.21) and (35.22) gives

$$w(C) \leq 2z^* \leq 2w(C^*) ,$$

and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. ∎

Exercises

35.4-1

Show that even if we allow a clause to contain both a variable and its negation, randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ still yields a randomized $8/7$-approximation algorithm.

35.4-2

The ***MAX-CNF satisfiability problem*** is like the MAX-3-CNF satisfiability problem, except that it does not restrict each clause to have exactly 3 literals. Give a randomized 2-approximation algorithm for the MAX-CNF satisfiability problem.

35.4-3

In the MAX-CUT problem, we are given an unweighted undirected graph $G = (V, E)$. We define a cut $(S, V - S)$ as in Chapter 23 and the ***weight*** of a cut as the number of edges crossing the cut. The goal is to find a cut of maximum weight. Suppose that for each vertex v, we randomly and independently place v in S with probability $1/2$ and in $V - S$ with probability $1/2$. Show that this algorithm is a randomized 2-approximation algorithm.

35.4-4

Show that the constraints in line (35.19) are redundant in the sense that if we remove them from the linear program in lines (35.17)–(35.20), any optimal solution to the resulting linear program must satisfy $x(v) \le 1$ for each $v \in V$.

35.5 The subset-sum problem

Recall from Section 34.5.5 that an instance of the subset-sum problem is a pair (S, t), where S is a set $\{x_1, x_2, \ldots, x_n\}$ of positive integers and t is a positive integer. This decision problem asks whether there exists a subset of S that adds up exactly to the target value t. As we saw in Section 34.5.5, this problem is NP-complete.

The optimization problem associated with this decision problem arises in practical applications. In the optimization problem, we wish to find a subset of $\{x_1, x_2, \ldots, x_n\}$ whose sum is as large as possible but not larger than t. For example, we may have a truck that can carry no more than t pounds, and n different boxes to ship, the ith of which weighs x_i pounds. We wish to fill the truck with as heavy a load as possible without exceeding the given weight limit.

In this section, we present an exponential-time algorithm that computes the optimal value for this optimization problem, and then we show how to modify the algorithm so that it becomes a fully polynomial-time approximation scheme. (Recall that a fully polynomial-time approximation scheme has a running time that is polynomial in $1/\epsilon$ as well as in the size of the input.)

An exponential-time exact algorithm

Suppose that we computed, for each subset S' of S, the sum of the elements in S', and then we selected, among the subsets whose sum does not exceed t, the one whose sum was closest to t. Clearly this algorithm would return the optimal solution, but it could take exponential time. To implement this algorithm, we could use an iterative procedure that, in iteration i, computes the sums of all subsets of $\{x_1, x_2, \ldots, x_i\}$, using as a starting point the sums of all subsets of $\{x_1, x_2, \ldots, x_{i-1}\}$. In doing so, we would realize that once a particular subset S' had a sum exceeding t, there would be no reason to maintain it, since no superset of S' could be the optimal solution. We now give an implementation of this strategy.

The procedure EXACT-SUBSET-SUM takes an input set $S = \{x_1, x_2, \ldots, x_n\}$ and a target value t; we'll see its pseudocode in a moment. This procedure iteratively computes L_i, the list of sums of all subsets of $\{x_1, \ldots, x_i\}$ that do not exceed t, and then it returns the maximum value in L_n.

If L is a list of positive integers and x is another positive integer, then we let $L + x$ denote the list of integers derived from L by increasing each element of L by x. For example, if $L = \langle 1, 2, 3, 5, 9 \rangle$, then $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. We also use this notation for sets, so that

$$S + x = \{s + x : s \in S\} \; .$$

We also use an auxiliary procedure MERGE-LISTS(L, L'), which returns the sorted list that is the merge of its two sorted input lists L and L' with duplicate values removed. Like the MERGE procedure we used in merge sort (Section 2.3.1), MERGE-LISTS runs in time $O(|L| + |L'|)$. We omit the pseudocode for MERGE-LISTS.

```
EXACT-SUBSET-SUM$(S, t)$
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  **for** $i = 1$ **to** $n$
4      $L_i =$ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
5      remove from $L_i$ every element that is greater than $t$
6  **return** the largest element in $L_n$
```

To see how EXACT-SUBSET-SUM works, let P_i denote the set of all values obtained by selecting a (possibly empty) subset of $\{x_1, x_2, \ldots, x_i\}$ and summing its members. For example, if $S = \{1, 4, 5\}$, then

$$\begin{aligned} P_1 &= \{0, 1\} \; , \\ P_2 &= \{0, 1, 4, 5\} \; , \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\} \; . \end{aligned}$$

Given the identity

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \; , \tag{35.23}$$

we can prove by induction on i (see Exercise 35.5-1) that the list L_i is a sorted list containing every element of P_i whose value is not more than t. Since the length of L_i can be as much as 2^i, EXACT-SUBSET-SUM is an exponential-time algorithm in general, although it is a polynomial-time algorithm in the special cases in which t is polynomial in $|S|$ or all the numbers in S are bounded by a polynomial in $|S|$.

A fully polynomial-time approximation scheme

We can derive a fully polynomial-time approximation scheme for the subset-sum problem by "trimming" each list L_i after it is created. The idea behind trimming is that if two values in L are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly. More precisely, we use a trimming parameter δ such that $0 < \delta < 1$. When we ***trim*** a list L by δ, we remove as many elements from L as possible, in such a way that if L' is the result of trimming L, then for every element y that was removed from L, there is an element z still in L' that approximates y, that is,

$$\frac{y}{1+\delta} \le z \le y \ . \tag{35.24}$$

We can think of such a z as "representing" y in the new list L'. Each removed element y is represented by a remaining element z satisfying inequality (35.24). For example, if $\delta = 0.1$ and

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \ ,$$

then we can trim L to obtain

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle \ ,$$

where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The following procedure trims list $L = \langle y_1, y_2, \ldots, y_m \rangle$ in time $\Theta(m)$, given L and δ, and assuming that L is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.

```
TRIM($L, \delta$)
1  let $m$ be the length of $L$
2  $L' = \langle y_1 \rangle$
3  $last = y_1$
4  for $i = 2$ to $m$
5      if $y_i > last \cdot (1 + \delta)$        // $y_i \ge last$ because $L$ is sorted
6          append $y_i$ onto the end of $L'$
7          $last = y_i$
8  return $L'$
```

The procedure scans the elements of L in monotonically increasing order. A number is appended onto the returned list L' only if it is the first element of L or if it cannot be represented by the most recent number placed into L'.

Given the procedure TRIM, we can construct our approximation scheme as follows. This procedure takes as input a set $S = \{x_1, x_2, \ldots, x_n\}$ of n integers (in arbitrary order), a target integer t, and an "approximation parameter" ϵ, where

$$0 < \epsilon < 1 . \tag{35.25}$$

It returns a value z whose value is within a $1 + \epsilon$ factor of the optimal solution.

```
APPROX-SUBSET-SUM($S, t, \epsilon$)
1  $n = |S|$
2  $L_0 = \langle 0 \rangle$
3  for $i = 1$ to $n$
4      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
5      $L_i$ = TRIM($L_i, \epsilon/2n$)
6      remove from $L_i$ every element that is greater than $t$
7  let $z^*$ be the largest value in $L_n$
8  return $z^*$
```

Line 2 initializes the list L_0 to be the list containing just the element 0. The **for** loop in lines 3–6 computes L_i as a sorted list containing a suitably trimmed version of the set P_i, with all elements larger than t removed. Since we create L_i from L_{i-1}, we must ensure that the repeated trimming doesn't introduce too much compounded inaccuracy. In a moment, we shall see that APPROX-SUBSET-SUM returns a correct approximation if one exists.

As an example, suppose we have the instance

$$S = \langle 104, 102, 201, 101 \rangle$$

with $t = 308$ and $\epsilon = 0.40$. The trimming parameter δ is $\epsilon/8 = 0.05$. APPROX-SUBSET-SUM computes the following values on the indicated lines:

$$\begin{aligned}
\text{line 2:} \quad L_0 &= \langle 0 \rangle , \\[6pt]
\text{line 4:} \quad L_1 &= \langle 0, 104 \rangle , \\
\text{line 5:} \quad L_1 &= \langle 0, 104 \rangle , \\
\text{line 6:} \quad L_1 &= \langle 0, 104 \rangle , \\[6pt]
\text{line 4:} \quad L_2 &= \langle 0, 102, 104, 206 \rangle , \\
\text{line 5:} \quad L_2 &= \langle 0, 102, 206 \rangle , \\
\text{line 6:} \quad L_2 &= \langle 0, 102, 206 \rangle , \\[6pt]
\text{line 4:} \quad L_3 &= \langle 0, 102, 201, 206, 303, 407 \rangle , \\
\text{line 5:} \quad L_3 &= \langle 0, 102, 201, 303, 407 \rangle , \\
\text{line 6:} \quad L_3 &= \langle 0, 102, 201, 303 \rangle , \\[6pt]
\text{line 4:} \quad L_4 &= \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle , \\
\text{line 5:} \quad L_4 &= \langle 0, 101, 201, 302, 404 \rangle , \\
\text{line 6:} \quad L_4 &= \langle 0, 101, 201, 302 \rangle .
\end{aligned}$$

The algorithm returns $z^* = 302$ as its answer, which is well within $\epsilon = 40\%$ of the optimal answer $307 = 104 + 102 + 101$; in fact, it is within 2%.

Theorem 35.8
APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

Proof The operations of trimming L_i in line 5 and removing from L_i every element that is greater than t maintain the property that every element of L_i is also a member of P_i. Therefore, the value z^* returned in line 8 is indeed the sum of some subset of S. Let $y^* \in P_n$ denote an optimal solution to the subset-sum problem. Then, from line 6, we know that $z^* \le y^*$. By inequality (35.1), we need to show that $y^*/z^* \le 1 + \epsilon$. We must also show that the running time of this algorithm is polynomial in both $1/\epsilon$ and the size of the input.

As Exercise 35.5-2 asks you to show, for every element y in P_i that is at most t, there exists an element $z \in L_i$ such that

$$\frac{y}{(1+\epsilon/2n)^i} \le z \le y . \tag{35.26}$$

Inequality (35.26) must hold for $y^* \in P_n$, and therefore there exists an element $z \in L_n$ such that

$$\frac{y^*}{(1+\epsilon/2n)^n} \le z \le y^* ,$$

and thus

$$\frac{y^*}{z} \le \left(1 + \frac{\epsilon}{2n}\right)^n . \tag{35.27}$$

Since there exists an element $z \in L_n$ fulfilling inequality (35.27), the inequality must hold for z^*, which is the largest value in L_n; that is,

$$\frac{y^*}{z^*} \le \left(1 + \frac{\epsilon}{2n}\right)^n . \tag{35.28}$$

Now, we show that $y^*/z^* \le 1 + \epsilon$. We do so by showing that $(1+\epsilon/2n)^n \le 1 + \epsilon$. By equation (3.14), we have $\lim_{n\to\infty}(1+\epsilon/2n)^n = e^{\epsilon/2}$. Exercise 35.5-3 asks you to show that

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0 . \tag{35.29}$$

Therefore, the function $(1+\epsilon/2n)^n$ increases with n as it approaches its limit of $e^{\epsilon/2}$, and we have

$$\begin{aligned}
\left(1 + \frac{\epsilon}{2n}\right)^n &\le e^{\epsilon/2} \\
&\le 1 + \epsilon/2 + (\epsilon/2)^2 && \text{(by inequality (3.13))} \\
&\le 1 + \epsilon && \text{(by inequality (35.25))} . \qquad (35.30)
\end{aligned}$$

Combining inequalities (35.28) and (35.30) completes the analysis of the approximation ratio.

To show that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme, we derive a bound on the length of L_i. After trimming, successive elements z and z' of L_i must have the relationship $z'/z > 1+\epsilon/2n$. That is, they must differ by a factor of at least $1 + \epsilon/2n$. Each list, therefore, contains the value 0, possibly the value 1, and up to $\lfloor \log_{1+\epsilon/2n} t \rfloor$ additional values. The number of elements in each list L_i is at most

$$\begin{aligned}
\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1 + \epsilon/2n)} + 2 \\
&\le \frac{2n(1 + \epsilon/2n) \ln t}{\epsilon} + 2 && \text{(by inequality (3.17))} \\
&< \frac{3n \ln t}{\epsilon} + 2 && \text{(by inequality (35.25))} .
\end{aligned}$$

This bound is polynomial in the size of the input—which is the number of bits $\lg t$ needed to represent t plus the number of bits needed to represent the set S, which is in turn polynomial in n—and in $1/\epsilon$. Since the running time of APPROX-SUBSET-SUM is polynomial in the lengths of the L_i, we conclude that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme. ■

Exercises

35.5-1

Prove equation (35.23). Then show that after executing line 5 of EXACT-SUBSET-SUM, L_i is a sorted list containing every element of P_i whose value is not more than t.

35.5-2

Using induction on i, prove inequality (35.26).

35.5-3

Prove inequality (35.29).

35.5-4

How would you modify the approximation scheme presented in this section to find a good approximation to the smallest value not less than t that is a sum of some subset of the given input list?

35.5-5

Modify the APPROX-SUBSET-SUM procedure to also return the subset of S that sums to the value z^*.

Problems

35-1 Bin packing

Suppose that we are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$. We wish to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

a. Prove that the problem of determining the minimum number of bins required is NP-hard. (*Hint:* Reduce from the subset-sum problem.)

The ***first-fit*** heuristic takes each object in turn and places it into the first bin that can accommodate it. Let $S = \sum_{i=1}^{n} s_i$.

b. Argue that the optimal number of bins required is at least $\lceil S \rceil$.

c. Argue that the first-fit heuristic leaves at most one bin less than half full.

d. Prove that the number of bins used by the first-fit heuristic is never more than $\lceil 2S \rceil$.

e. Prove an approximation ratio of 2 for the first-fit heuristic.

f. Give an efficient implementation of the first-fit heuristic, and analyze its running time.

35-2 Approximating the size of a maximum clique

Let $G = (V, E)$ be an undirected graph. For any $k \ge 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k-tuples of vertices from V and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for $i = 1, 2, \ldots, k$, either vertex v_i is adjacent to w_i in G, or else $v_i = w_i$.

a. Prove that the size of the maximum clique in $G^{(k)}$ is equal to the kth power of the size of the maximum clique in G.

b. Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a polynomial-time approximation scheme for the problem.

35-3 Weighted set-covering problem

Suppose that we generalize the set-covering problem so that each set S_i in the family $\mathcal{F}$ has an associated weight w_i and the weight of a cover $\mathcal{C}$ is $\sum_{S_i \in \mathcal{C}} w_i$. We wish to determine a minimum-weight cover. (Section 35.3 handles the case in which $w_i = 1$ for all i.)

Show how to generalize the greedy set-covering heuristic in a natural manner to provide an approximate solution for any instance of the weighted set-covering problem. Show that your heuristic has an approximation ratio of $H(d)$, where d is the maximum size of any set S_i.

35-4 Maximum matching

Recall that for an undirected graph G, a matching is a set of edges such that no two edges in the set are incident on the same vertex. In Section 26.3, we saw how to find a maximum matching in a bipartite graph. In this problem, we will look at matchings in undirected graphs in general (i.e., the graphs are not required to be bipartite).

a. A ***maximal matching*** is a matching that is not a proper subset of any other matching. Show that a maximal matching need not be a maximum matching by exhibiting an undirected graph G and a maximal matching M in G that is not a maximum matching. (*Hint:* You can find such a graph with only four vertices.)

b. Consider an undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in G.

In this problem, we shall concentrate on a polynomial-time approximation algorithm for maximum matching. Whereas the fastest known algorithm for maximum matching takes superlinear (but polynomial) time, the approximation algorithm here will run in linear time. You will show that the linear-time greedy algorithm for maximal matching in part (b) is a 2-approximation algorithm for maximum matching.

c. Show that the size of a maximum matching in G is a lower bound on the size of any vertex cover for G.

d. Consider a maximal matching M in $G = (V, E)$. Let

$$T = \{v \in V : \text{some edge in } M \text{ is incident on } v\} .$$

What can you say about the subgraph of G induced by the vertices of G that are not in T?

e. Conclude from part (d) that $2|M|$ is the size of a vertex cover for G.

f. Using parts (c) and (e), prove that the greedy algorithm in part (b) is a 2-approximation algorithm for maximum matching.

35-5 Parallel machine scheduling

In the ***parallel-machine-scheduling problem***, we are given n jobs, $J_1, J_2, \ldots, J_n$, where each job J_k has an associated nonnegative processing time of p_k. We are also given m identical machines, $M_1, M_2, \ldots, M_m$. Any job can run on any machine. A ***schedule*** specifies, for each job J_k, the machine on which it runs and the time period during which it runs. Each job J_k must run on some machine M_i for p_k consecutive time units, and during that time period no other job may run on M_i. Let C_k denote the ***completion time*** of job J_k, that is, the time at which job J_k completes processing. Given a schedule, we define $C_{\max} = \max_{1 \le j \le n} C_j$ to be the ***makespan*** of the schedule. The goal is to find a schedule whose makespan is minimum.

For example, suppose that we have two machines M_1 and M_2 and that we have four jobs J_1, J_2, J_3, J_4, with $p_1 = 2$, $p_2 = 12$, $p_3 = 4$, and $p_4 = 5$. Then one possible schedule runs, on machine M_1, job J_1 followed by job J_2, and on machine M_2, it runs job J_4 followed by job J_3. For this schedule, $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$, and $C_{\max} = 14$. An optimal schedule runs J_2 on machine M_1, and it runs jobs J_1, J_3, and J_4 on machine M_2. For this schedule, $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, $C_4 = 11$, and $C_{\max} = 12$.

Given a parallel-machine-scheduling problem, we let $C^*_{\max}$ denote the makespan of an optimal schedule.

a. Show that the optimal makespan is at least as large as the greatest processing time, that is,

$$C^*_{\max} \ge \max_{1 \le k \le n} p_k .$$

b. Show that the optimal makespan is at least as large as the average machine load, that is,

$$C^*_{\max} \ge \frac{1}{m} \sum_{1 \le k \le n} p_k .$$

Suppose that we use the following greedy algorithm for parallel machine scheduling: whenever a machine is idle, schedule any job that has not yet been scheduled.

c. Write pseudocode to implement this greedy algorithm. What is the running time of your algorithm?

d. For the schedule returned by the greedy algorithm, show that

$$C_{\max} \le \frac{1}{m} \sum_{1 \le k \le n} p_k + \max_{1 \le k \le n} p_k \; .$$

Conclude that this algorithm is a polynomial-time 2-approximation algorithm.

35-6 *Approximating a maximum spanning tree*

Let $G = (V, E)$ be an undirected graph with distinct edge weights $w(u, v)$ on each edge $(u, v) \in E$. For each vertex $v \in V$, let $\max(v) = \max_{(u,v) \in E} \{w(u, v)\}$ be the maximum-weight edge incident on that vertex. Let $S_G = \{\max(v) : v \in V\}$ be the set of maximum-weight edges incident on each vertex, and let T_G be the maximum-weight spanning tree of G, that is, the spanning tree of maximum total weight. For any subset of edges $E' \subseteq E$, define $w(E') = \sum_{(u,v) \in E'} w(u, v)$.

a. Give an example of a graph with at least 4 vertices for which $S_G = T_G$.

b. Give an example of a graph with at least 4 vertices for which $S_G \ne T_G$.

c. Prove that $S_G \subseteq T_G$ for any graph G.

d. Prove that $w(T_G) \ge w(S_G)/2$ for any graph G.

e. Give an $O(V + E)$-time algorithm to compute a 2-approximation to the maximum spanning tree.

35-7 *An approximation algorithm for the 0-1 knapsack problem*

Recall the knapsack problem from Section 16.2. There are n items, where the ith item is worth v_i dollars and weighs w_i pounds. We are also given a knapsack that can hold at most W pounds. Here, we add the further assumptions that each weight w_i is at most W and that the items are indexed in monotonically decreasing order of their values: $v_1 \ge v_2 \ge \cdots \ge v_n$.

In the 0-1 knapsack problem, we wish to find a subset of the items whose total weight is at most W and whose total value is maximum. The fractional knapsack problem is like the 0-1 knapsack problem, except that we are allowed to take a fraction of each item, rather than being restricted to taking either all or none of each item. If we take a fraction x_i of item i, where $0 \le x_i \le 1$, we contribute $x_i w_i$ to the weight of the knapsack and receive value $x_i v_i$. Our goal is to develop a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, we consider restricted instances of the 0-1 knapsack problem. Given an instance I of the knapsack problem, we form restricted instances I_j, for $j = 1, 2, \ldots, n$, by removing items $1, 2, \ldots, j-1$ and requiring the solution to include item j (all of item j in both the fractional and 0-1 knapsack problems). No items are removed in instance I_1. For instance I_j, let P_j denote an optimal solution to the 0-1 problem and Q_j denote an optimal solution to the fractional problem.

a. Argue that an optimal solution to instance I of the 0-1 knapsack problem is one of $\{P_1, P_2, \ldots, P_n\}$.

b. Prove that we can find an optimal solution Q_j to the fractional problem for instance I_j by including item j and then using the greedy algorithm in which at each step, we take as much as possible of the unchosen item in the set $\{j+1, j+2, \ldots, n\}$ with maximum value per pound v_i/w_i.

c. Prove that we can always construct an optimal solution Q_j to the fractional problem for instance I_j that includes at most one item fractionally. That is, for all items except possibly one, we either include all of the item or none of the item in the knapsack.

d. Given an optimal solution Q_j to the fractional problem for instance I_j, form solution R_j from Q_j by deleting any fractional items from Q_j. Let $v(S)$ denote the total value of items taken in a solution S. Prove that $v(R_j) \ge v(Q_j)/2 \ge v(P_j)/2$.

e. Give a polynomial-time algorithm that returns a maximum-value solution from the set $\{R_1, R_2, \ldots, R_n\}$, and prove that your algorithm is a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

Chapter notes

Although methods that do not necessarily compute exact solutions have been known for thousands of years (for example, methods to approximate the value of π), the notion of an approximation algorithm is much more recent. Hochbaum [172] credits Garey, Graham, and Ullman [128] and Johnson [190] with formalizing the concept of a polynomial-time approximation algorithm. The first such algorithm is often credited to Graham [149].

Since this early work, thousands of approximation algorithms have been designed for a wide range of problems, and there is a wealth of literature on this field. Recent texts by Ausiello et al. [26], Hochbaum [172], and Vazirani [345] deal exclusively with approximation algorithms, as do surveys by Shmoys [315] and Klein and Young [207]. Several other texts, such as Garey and Johnson [129] and Papadimitriou and Steiglitz [271], have significant coverage of approximation algorithms as well. Lawler, Lenstra, Rinnooy Kan, and Shmoys [225] provide an extensive treatment of approximation algorithms for the traveling-salesman problem.

Papadimitriou and Steiglitz attribute the algorithm APPROX-VERTEX-COVER to F. Gavril and M. Yannakakis. The vertex-cover problem has been studied extensively (Hochbaum [172] lists 16 different approximation algorithms for this problem), but all the approximation ratios are at least $2 - o(1)$.

The algorithm APPROX-TSP-TOUR appears in a paper by Rosenkrantz, Stearns, and Lewis [298]. Christofides improved on this algorithm and gave a $3/2$-approximation algorithm for the traveling-salesman problem with the triangle inequality. Arora [22] and Mitchell [257] have shown that if the points are in the euclidean plane, there is a polynomial-time approximation scheme. Theorem 35.3 is due to Sahni and Gonzalez [301].

The analysis of the greedy heuristic for the set-covering problem is modeled after the proof published by Chvátal [68] of a more general result; the basic result as presented here is due to Johnson [190] and Lovász [238].

The algorithm APPROX-SUBSET-SUM and its analysis are loosely modeled after related approximation algorithms for the knapsack and subset-sum problems by Ibarra and Kim [187].

Problem 35-7 is a combinatorial version of a more general result on approximating knapsack-type integer programs by Bienstock and McClosky [45].

The randomized algorithm for MAX-3-CNF satisfiability is implicit in the work of Johnson [190]. The weighted vertex-cover algorithm is by Hochbaum [171]. Section 35.4 only touches on the power of randomization and linear programming in the design of approximation algorithms. A combination of these two ideas yields a technique called "randomized rounding," which formulates a problem as an integer linear program, solves the linear-programming relaxation, and interprets the variables in the solution as probabilities. These probabilities then help guide the solution of the original problem. This technique was first used by Raghavan and Thompson [290], and it has had many subsequent uses. (See Motwani, Naor, and Raghavan [261] for a survey.) Several other notable recent ideas in the field of approximation algorithms include the primal-dual method (see Goemans and Williamson [135] for a survey), finding sparse cuts for use in divide-and-conquer algorithms [229], and the use of semidefinite programming [134].

As mentioned in the chapter notes for Chapter 34, recent results in probabilistically checkable proofs have led to lower bounds on the approximability of many problems, including several in this chapter. In addition to the references there, the chapter by Arora and Lund [23] contains a good description of the relationship between probabilistically checkable proofs and the hardness of approximating various problems.