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

Introduction to Algorithms

Third Edition

I Foundations

Introduction

This part will start you thinking about designing and analyzing algorithms. It is intended to be a gentle introduction to how we specify algorithms, some of the design strategies we will use throughout this book, and many of the fundamental ideas used in algorithm analysis. Later parts of this book will build upon this base.

Chapter 1 provides an overview of algorithms and their place in modern computing systems. This chapter defines what an algorithm is and lists some examples. It also makes a case that we should consider algorithms as a technology, alongside technologies such as fast hardware, graphical user interfaces, object-oriented systems, and networks.

In Chapter 2, we see our first algorithms, which solve the problem of sorting a sequence of n numbers. They are written in a pseudocode which, although not directly translatable to any conventional programming language, conveys the structure of the algorithm clearly enough that you should be able to implement it in the language of your choice. The sorting algorithms we examine are insertion sort, which uses an incremental approach, and merge sort, which uses a recursive technique known as "divide-and-conquer." Although the time each requires increases with the value of n, the rate of increase differs between the two algorithms. We determine these running times in Chapter 2, and we develop a useful notation to express them.

Chapter 3 precisely defines this notation, which we call asymptotic notation. It starts by defining several asymptotic notations, which we use for bounding algorithm running times from above and/or below. The rest of Chapter 3 is primarily a presentation of mathematical notation, more to ensure that your use of notation matches that in this book than to teach you new mathematical concepts.

Chapter 4 delves further into the divide-and-conquer method introduced in Chapter 2. It provides additional examples of divide-and-conquer algorithms, including Strassen's surprising method for multiplying two square matrices. Chapter 4 contains methods for solving recurrences, which are useful for describing the running times of recursive algorithms. One powerful technique is the "master method," which we often use to solve recurrences that arise from divide-and-conquer algorithms. Although much of Chapter 4 is devoted to proving the correctness of the master method, you may skip this proof yet still employ the master method.

Chapter 5 introduces probabilistic analysis and randomized algorithms. We typically use probabilistic analysis to determine the running time of an algorithm in cases in which, due to the presence of an inherent probability distribution, the running time may differ on different inputs of the same size. In some cases, we assume that the inputs conform to a known probability distribution, so that we are averaging the running time over all possible inputs. In other cases, the probability distribution comes not from the inputs but from random choices made during the course of the algorithm. An algorithm whose behavior is determined not only by its input but by the values produced by a random-number generator is a randomized algorithm. We can use randomized algorithms to enforce a probability distribution on the inputs—thereby ensuring that no particular input always causes poor performance—or even to bound the error rate of algorithms that are allowed to produce incorrect results on a limited basis.

Appendices A–D contain other mathematical material that you will find helpful as you read this book. You are likely to have seen much of the material in the appendix chapters before having read this book (although the specific definitions and notational conventions we use may differ in some cases from what you have seen in the past), and so you should think of the Appendices as reference material. On the other hand, you probably have not already seen most of the material in Part I. All the chapters in Part I and the Appendices are written with a tutorial flavor.

1 The Role of Algorithms in Computing

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers? In this chapter, we will answer these questions.

1.1 Algorithms

Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

For example, we might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the ***sorting problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

For example, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$. Such an input sequence is called an ***instance*** of the sorting problem. In general, an ***instance of a problem*** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, we have a large number of good sorting algorithms at our disposal. Which algorithm is best for a given application depends on—among other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.

An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output. We say that a correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate. We shall see an example of an algorithm with a controllable error rate in Chapter 31 when we study algorithms for finding large prime numbers. Ordinarily, however, we shall be concerned only with correct algorithms.

An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

What kinds of problems are solved by algorithms?

Sorting is by no means the only computational problem for which algorithms have been developed. (You probably suspected as much when you saw the size of this book.) Practical applications of algorithms are ubiquitous and include the following examples:

- The Human Genome Project has made great progress toward the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. Although the solutions to the various problems involved are beyond the scope of this book, many methods to solve these biological problems use ideas from several of the chapters in this book, thereby enabling scientists to accomplish tasks while using resources efficiently. The savings are in time, both human and machine, and in money, as more information can be extracted from laboratory techniques.
- The Internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the Internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data will travel (techniques for solving such problems appear in

Chapter 24), and using a search engine to quickly find pages on which particular information resides (related techniques are in Chapters 11 and 32).

- Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements. The core technologies used in electronic commerce include public-key cryptography and digital signatures (covered in Chapter 31), which are based on numerical algorithms and number theory.
- Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met. An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming, which we shall study in Chapter 29.

Although some of the details of these examples are beyond the scope of this book, we do give underlying techniques that apply to these problems and problem areas. We also show how to solve many specific problems, including the following:

- We are given a road map on which the distance between each pair of adjacent intersections is marked, and we wish to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if we disallow routes that cross over themselves. How do we choose which of all possible routes is the shortest? Here, we model the road map (which is itself a model of the actual roads) as a graph (which we will meet in Part VI and Appendix B), and we wish to find the shortest path from one vertex to another in the graph. We shall see how to solve this problem efficiently in Chapter 24.
- We are given two ordered sequences of symbols, $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, and we wish to find a longest common subsequence of X and Y. A subsequence of X is just X with some (or possibly all or none) of its elements removed. For example, one subsequence of $\langle A, B, C, D, E, F, G \rangle$ would be $\langle B, C, E, G \rangle$. The length of a longest common subsequence of X and Y gives one measure of how similar these two sequences are. For example, if the two sequences are base pairs in DNA strands, then we might consider them similar if they have a long common subsequence. If X has m symbols and Y has n symbols, then X and Y have 2^m and 2^n possible subsequences,

respectively. Selecting all possible subsequences of X and Y and matching them up could take a prohibitively long time unless m and n are very small. We shall see in Chapter 15 how to use a general technique known as dynamic programming to solve this problem much more efficiently.

- We are given a mechanical design in terms of a library of parts, where each part may include instances of other parts, and we need to list the parts in order so that each part appears before any part that uses it. If the design comprises n parts, then there are $n!$ possible orders, where $n!$ denotes the factorial function. Because the factorial function grows faster than even an exponential function, we cannot feasibly generate each possible order and then verify that, within that order, each part appears before the parts using it (unless we have only a few parts). This problem is an instance of topological sorting, and we shall see in Chapter 22 how to solve this problem efficiently.
- We are given n points in the plane, and we wish to find the convex hull of these points. The convex hull is the smallest convex polygon containing the points. Intuitively, we can think of each point as being represented by a nail sticking out from a board. The convex hull would be represented by a tight rubber band that surrounds all the nails. Each nail around which the rubber band makes a turn is a vertex of the convex hull. (See Figure 33.6 on page 1029 for an example.) Any of the 2^n subsets of the points might be the vertices of the convex hull. Knowing which points are vertices of the convex hull is not quite enough, either, since we also need to know the order in which they appear. There are many choices, therefore, for the vertices of the convex hull. Chapter 33 gives two good methods for finding the convex hull.

These lists are far from exhaustive (as you again have probably surmised from this book's heft), but exhibit two characteristics that are common to many interesting algorithmic problems:

1. They have many candidate solutions, the overwhelming majority of which do not solve the problem at hand. Finding one that does, or one that is "best," can present quite a challenge.
2. They have practical applications. Of the problems in the above list, finding the shortest path provides the easiest examples. A transportation firm, such as a trucking or railroad company, has a financial interest in finding shortest paths through a road or rail network because taking shorter paths results in lower labor and fuel costs. Or a routing node on the Internet may need to find the shortest path through the network in order to route a message quickly. Or a person wishing to drive from New York to Boston may want to find driving directions from an appropriate Web site, or she may use her GPS while driving.

Not every problem solved by algorithms has an easily identified set of candidate solutions. For example, suppose we are given a set of numerical values representing samples of a signal, and we want to compute the discrete Fourier transform of these samples. The discrete Fourier transform converts the time domain to the frequency domain, producing a set of numerical coefficients, so that we can determine the strength of various frequencies in the sampled signal. In addition to lying at the heart of signal processing, discrete Fourier transforms have applications in data compression and multiplying large polynomials and integers. Chapter 30 gives an efficient algorithm, the fast Fourier transform (commonly called the FFT), for this problem, and the chapter also sketches out the design of a hardware circuit to compute the FFT.

Data structures

This book also contains several data structures. A ***data structure*** is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

Technique

Although you can use this book as a "cookbook" for algorithms, you may someday encounter a problem for which you cannot readily find a published algorithm (many of the exercises and problems in this book, for example). This book will teach you techniques of algorithm design and analysis so that you can develop algorithms on your own, show that they give the correct answer, and understand their efficiency. Different chapters address different aspects of algorithmic problem solving. Some chapters address specific problems, such as finding medians and order statistics in Chapter 9, computing minimum spanning trees in Chapter 23, and determining a maximum flow in a network in Chapter 26. Other chapters address techniques, such as divide-and-conquer in Chapter 4, dynamic programming in Chapter 15, and amortized analysis in Chapter 17.

Hard problems

Most of this book is about efficient algorithms. Our usual measure of efficiency is speed, i.e., how long an algorithm takes to produce its result. There are some problems, however, for which no efficient solution is known. Chapter 34 studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven

that an efficient algorithm for one cannot exist. In other words, no one knows whether or not efficient algorithms exist for NP-complete problems. Second, the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. Computer scientists are intrigued by how a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

You should know about NP-complete problems because some of them arise surprisingly often in real applications. If you are called upon to produce an efficient algorithm for an NP-complete problem, you are likely to spend a lot of time in a fruitless search. If you can show that the problem is NP-complete, you can instead spend your time developing an efficient algorithm that gives a good, but not the best possible, solution.

As a concrete example, consider a delivery company with a central depot. Each day, it loads up each delivery truck at the depot and sends it around to deliver goods to several addresses. At the end of the day, each truck must end up back at the depot so that it is ready to be loaded for the next day. To reduce costs, the company wants to select an order of delivery stops that yields the lowest overall distance traveled by each truck. This problem is the well-known "traveling-salesman problem," and it is NP-complete. It has no known efficient algorithm. Under certain assumptions, however, we know of efficient algorithms that give an overall distance which is not too far above the smallest possible. Chapter 35 discusses such "approximation algorithms."

Parallelism

For many years, we could count on processor clock speeds increasing at a steady rate. Physical limitations present a fundamental roadblock to ever-increasing clock speeds, however: because power density increases superlinearly with clock speed, chips run the risk of melting once their clock speeds become high enough. In order to perform more computations per second, therefore, chips are being designed to contain not just one but several processing "cores." We can liken these multicore computers to several sequential computers on a single chip; in other words, they are a type of "parallel computer." In order to elicit the best performance from multicore computers, we need to design algorithms with parallelism in mind. Chapter 27 presents a model for "multithreaded" algorithms, which take advantage of multiple cores. This model has advantages from a theoretical standpoint, and it forms the basis of several successful computer programs, including a championship chess program.

Exercises

1.1-1
Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.

1.1-2
Other than speed, what other measures of efficiency might one use in a real-world setting?

1.1-3
Select a data structure that you have seen previously, and discuss its strengths and limitations.

1.1-4
How are the shortest-path and traveling-salesman problems given above similar? How are they different?

1.1-5
Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

1.2 Algorithms as a technology

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement.

Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

Efficiency

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $50n \lg n$ instructions. To sort 10 million numbers, computer A takes

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)} ,$$

while computer B takes

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)} .$$

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of merge sort is even more pronounced when we sort 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. In general, as the problem size increases, so does the relative advantage of merge sort.

Algorithms and other technologies

The example above shows that we should consider algorithms, like computer hardware, as a ***technology***. Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

You might wonder whether algorithms are truly that important on contemporary computers in light of other advanced technologies, such as

- advanced computer architectures and fabrication technologies,
- easy-to-use, intuitive, graphical user interfaces (GUIs),
- object-oriented systems,
- integrated Web technologies, and
- fast networking, both wired and wireless.

The answer is yes. Although some applications do not explicitly require algorithmic content at the application level (such as some simple, Web-based applications), many do. For example, consider a Web-based service that determines how to travel from one location to another. Its implementation would rely on fast hardware, a graphical user interface, wide-area networking, and also possibly on object orientation. However, it would also require algorithms for certain operations, such as finding routes (probably using a shortest-path algorithm), rendering maps, and interpolating addresses.

Moreover, even an application that does not require algorithmic content at the application level relies heavily upon algorithms. Does the application rely on fast hardware? The hardware design used algorithms. Does the application rely on graphical user interfaces? The design of any GUI relies on algorithms. Does the application rely on networking? Routing in networks relies heavily on algorithms. Was the application written in a language other than machine code? Then it was processed by a compiler, interpreter, or assembler, all of which make extensive use

of algorithms. Algorithms are at the core of most technologies used in contemporary computers.

Furthermore, with the ever-increasing capacities of computers, we use them to solve larger problems than ever before. As we saw in the above comparison between insertion sort and merge sort, it is at larger problem sizes that the differences in efficiency between algorithms become particularly prominent.

Having a solid base of algorithmic knowledge and technique is one characteristic that separates the truly skilled programmers from the novices. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more.

Exercises

1.2-1
Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

1.2-2
Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

1.2-3
What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Problems

1-1 Comparison of running times
For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Chapter notes

There are many excellent texts on the general topic of algorithms, including those by Aho, Hopcroft, and Ullman [5, 6]; Baase and Van Gelder [28]; Brassard and Bratley [54]; Dasgupta, Papadimitriou, and Vazirani [82]; Goodrich and Tamassia [148]; Hofri [175]; Horowitz, Sahni, and Rajasekaran [181]; Johnsonbaugh and Schaefer [193]; Kingston [205]; Kleinberg and Tardos [208]; Knuth [209, 210, 211]; Kozen [220]; Levitin [235]; Manber [242]; Mehlhorn [249, 250, 251]; Purdom and Brown [287]; Reingold, Nievergelt, and Deo [293]; Sedgewick [306]; Sedgewick and Flajolet [307]; Skiena [318]; and Wilf [356]. Some of the more practical aspects of algorithm design are discussed by Bentley [42, 43] and Gonnet [145]. Surveys of the field of algorithms can also be found in the *Handbook of Theoretical Computer Science, Volume A* [342] and the CRC *Algorithms and Theory of Computation Handbook* [25]. Overviews of the algorithms used in computational biology can be found in textbooks by Gusfield [156], Pevzner [275], Setubal and Meidanis [310], and Waterman [350].

2 Getting Started

This chapter will familiarize you with the framework we shall use throughout the book to think about the design and analysis of algorithms. It is self-contained, but it does include several references to material that we introduce in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We define a "pseudocode" that should be familiar to you if you have done computer programming, and we use it to show how we shall specify our algorithms. Having specified the insertion sort algorithm, we then argue that it correctly sorts, and we analyze its running time. The analysis introduces a notation that focuses on how that time increases with the number of items to be sorted. Following our discussion of insertion sort, we introduce the divide-and-conquer approach to the design of algorithms and use it to develop an algorithm called merge sort. We end with an analysis of merge sort's running time.

2.1 Insertion sort

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in Chapter 1:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The numbers that we wish to sort are also known as the ***keys***. Although conceptually we are sorting a sequence, the input comes to us in the form of an array with n elements.

In this book, we shall typically describe algorithms as programs written in a ***pseudocode*** that is similar in many respects to C, C++, Java, Python, or Pascal. If you have been introduced to any of these languages, you should have little trouble

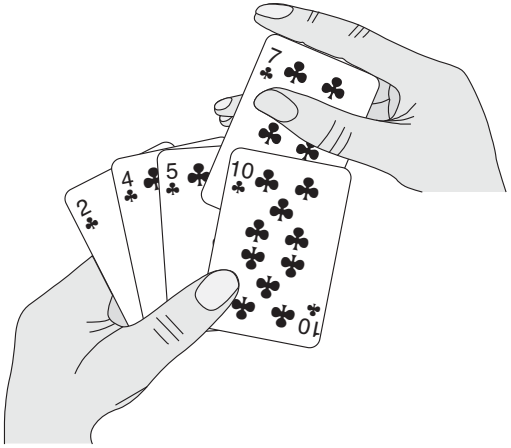


Figure 2.1 Sorting a hand of cards using insertion sort.

reading our algorithms. What separates pseudocode from "real" code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of "real" code. Another difference between pseudocode and real code is that pseudocode is not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

We present our pseudocode for insertion sort as a procedure called INSERTION-SORT, which takes as a parameter an array $A[1..n]$ containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by $A.length$.) The algorithm sorts the input numbers ***in place***: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished.

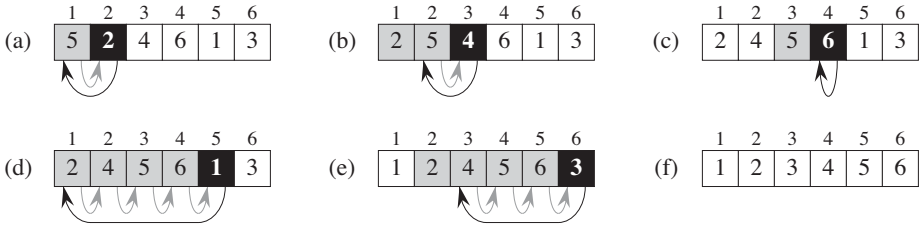


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by j, the subarray consisting of elements $A[1 \,..\, j-1]$ constitutes the currently sorted hand, and the remaining subarray $A[j+1 \,..\, n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 \,..\, j-1]$ are the elements *originally* in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \,..\, j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \,..\, j-1]$ consists of the elements originally in $A[1 \,..\, j-1]$, but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop. (Of course, we are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration.) Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

The third property is perhaps the most important one, since we are using the loop invariant to show correctness. Typically, we use the loop invariant along with the condition that caused the loop to terminate. The termination property differs from how we usually use mathematical induction, in which we apply the inductive step infinitely; here, we stop the "induction" when the loop terminates.

Let us see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1 \,..\, j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 \,..\, j]$ then consists of the elements originally in $A[1 \,..\, j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. At this point, however,

---

[1] When the loop is a **for** loop, the moment at which we check the loop invariant just prior to the first iteration is immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable j but before the first test of whether $j \leq A.length$.

we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this chapter and in other chapters as well.

Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if-else** statements[2] as well. Using indentation instead of conventional indicators of block structure, such as **begin** and **end** statements, greatly reduces clutter while preserving, or even enhancing, clarity.[3]
- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.[4] In this book, the loop counter retains its value after exiting the loop, unlike some situations that arise in C++, Java, and Pascal. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound. We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $j = 2$ **to** $A.length$, and so when this loop terminates, $j = A.length + 1$ (or, equivalently, $j = n + 1$, since $n = A.length$). We use the keyword **to** when a **for** loop increments its loop

---

[2]In an **if-else** statement, we indent **else** at the same level as its matching **if**. Although we omit the keyword **then**, we occasionally refer to the portion executed when the test following **if** is true as a ***then clause***. For multiway tests, we use **elseif** for tests after the first one.

[3]Each pseudocode procedure in this book appears on one page so that you will not have to discern levels of indentation in code that is split across pages.

[4]Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat-until** loops, and its **for** loops operate a little differently from the **for** loops in this book.

counter in each iteration, and we use the keyword **downto** when a **for** loop decrements its loop counter. When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.

- The symbol "**//**" indicates that the remainder of the line is a comment.
- A multiple assignment of the form $i = j = e$ assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment $j = e$ followed by the assignment $i = j$.
- Variables (such as i, j, and *key*) are local to the given procedure. We shall not use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the ith element of the array A. The notation ". ." is used to indicate a range of values within an array. Thus, $A[1 \,..\, j]$ indicates the subarray of A consisting of the j elements $A[1], A[2], \ldots, A[j]$.
- We typically organize compound data into ***objects***, which are composed of ***attributes***. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, we treat an array as an object with the attribute *length* indicating how many elements it contains. To specify the number of elements in an array A, we write $A.length$.

  We treat a variable representing an array or object as a pointer to the data representing the array or object. For all attributes f of an object x, setting $y = x$ causes $y.f$ to equal $x.f$. Moreover, if we now set $x.f = 3$, then afterward not only does $x.f$ equal 3, but $y.f$ equals 3 as well. In other words, x and y point to the same object after the assignment $y = x$.

  Our attribute notation can "cascade." For example, suppose that the attribute f is itself a pointer to some type of object that has an attribute g. Then the notation $x.f.g$ is implicitly parenthesized as $(x.f).g$. In other words, if we had assigned $y = x.f$, then $x.f.g$ is the same as $y.g$.

  Sometimes, a pointer will refer to no object at all. In this case, we give it the special value NIL.
- We pass parameters to a procedure ***by value***: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not. For example, if x is a parameter of a called procedure, the assignment $x = y$ within the called procedure is not visible to the calling procedure. The assignment $x.f = 3$, however, is visible. Similarly, arrays are passed by pointer, so that

a pointer to the array is passed, rather than the entire array, and changes to individual array elements are visible to the calling procedure.

- A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single **return** statement.
- The boolean operators "and" and "or" are ***short circuiting***. That is, when we evaluate the expression "x and y" we first evaluate x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and so we do not evaluate y. If, on the other hand, x evaluates to TRUE, we must evaluate y to determine the value of the entire expression. Similarly, in the expression "x or y" we evaluate the expression y only if x evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ NIL and $x.f = y$" without worrying about what happens when we try to evaluate $x.f$ when x is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called. The calling procedure is responsible for handling the error, and so we do not specify what action to take.

Exercises

2.1-1
Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2
Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

2.1-3
Consider the ***searching problem***:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.

Output: An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.

Write pseudocode for ***linear search***, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-4
Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in

an $(n + 1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one-processor, ***random-access machine (RAM)*** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size n, we typically assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—clearly an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant-time instruction? In the general case, no; it takes several instructions to compute x^y when x and y are real numbers. In restricted situations, however, exponentiation is a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by k positions to the left. In most computers, shifting the bits of an integer by one position to the left is equivalent to multiplication by 2, so that shifting the bits by k positions to the left is equivalent to multiplication by 2^k. Therefore, such computers can compute 2^k in one constant-time instruction by shifting the integer 1 by k positions to the left, as long as k is no more than the number of bits in a computer word. We will endeavor to avoid such gray areas in the RAM model, but we will treat computation of 2^k as a constant-time operation when k is a small enough positive integer.

In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory. Several computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. A handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book will not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Even though we typically select only one machine model to analyze a given algorithm, we still face many choices in deciding how to express our analysis. We would like a way that is simple to write and manipulate, shows the important characteristics of an algorithm's resource requirements, and suppresses tedious details.

Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully.

The best notion for ***input size*** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input*—for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time c_i, where c_i is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[5]

In the following discussion, our expression for the running time of INSERTION-SORT will evolve from a messy formula that uses all the statement costs c_i to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots, n$, where $n = A.length$, we let t_j denote the number of times the **while** loop test in line 5 is executed for that value of j. When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

---

[5]There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. For example, later in this book we might say "sort the points by x-coordinate," which, as we shall see, takes more than a constant amount of time. Also, note that a statement that calls a subroutine takes constant time, though the subroutine, once invoked, may take more. That is, we separate the process of ***calling*** the subroutine—passing parameters to it, etc.—from the process of ***executing*** the subroutine.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \; . \end{aligned}$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \le key$ in line 5 when i has its initial value of $j-1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \; . \end{aligned}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_i; it is thus a ***linear function*** of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

[6] This characteristic does not necessarily hold for a resource such as memory. A statement that references m words of memory and is executed n times does not necessarily reference mn distinct words of memory.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(see Appendix A for a review of how to solve these summations), we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a ***quadratic function*** of n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size n. We give three reasons for this orientation.

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.

- The "average case" is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray $A[1 \,..\, j-1]$ to insert element $A[j]$? On average, half the elements in $A[1 \,..\, j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1 \,..\, j-1]$, and so t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we shall be interested in the ***average-case*** running time of an algorithm; we shall see the technique of ***probabilistic analysis*** applied to various algorithms throughout this book. The scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem. Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a ***randomized algorithm***, which makes random choices, to allow a probabilistic analysis and yield an ***expected*** running time. We explore randomized algorithms more in Chapter 5 and in several other subsequent chapters.

Order of growth

We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure. First, we ignored the actual cost of each statement, using the constants c_i to represent these costs. Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i. We thus ignored not only the actual statement costs, but also the abstract costs c_i.

We shall now make one more simplifying abstraction: it is the ***rate of growth***, or ***order of growth***, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort, when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of n^2 from the leading term. We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared"). We shall use Θ-notation informally in this chapter, and we will define it precisely in Chapter 3.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower

order of growth. But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\Theta(n^3)$ algorithm.

Exercises

2.2-1
Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

2.2-2
Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

2.2-3
Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

2.2-4
How can we modify almost any algorithm to have a good best-case running time?

2.3 Designing algorithms

We can choose from a wide range of algorithm design techniques. For insertion sort, we used an ***incremental*** approach: having sorted the subarray $A[1 \mathinner{..} j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1 \mathinner{..} j]$.

In this section, we examine an alternative design approach, known as "divide-and-conquer," which we shall explore in more detail in Chapter 4. We'll use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined using techniques that we will see in Chapter 4.

2.3.1 The divide-and-conquer approach

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves 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.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices into the array such that $p \le q < r$. The procedure assumes that the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray $A[p\,..\,r]$.

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto

the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use ∞ as the sentinel value, so that whenever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p \,..\, q]$, and line 2 computes the length n_2 of the subarray $A[q + 1 \,..\, r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray $A[p \,..\, q]$ into $L[1 \,..\, n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q + 1 \,..\, r]$ into $R[1 \,..\, n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illus-

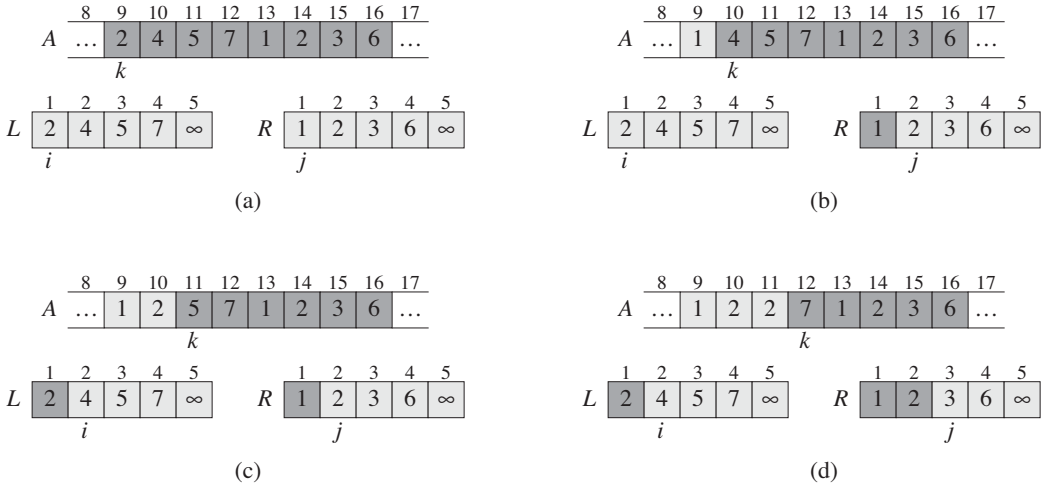


Figure 2.3 The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9\,..\,16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty\rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty\rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9\,..\,16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.

trated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

> At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p\,..\,k-1]$ contains the $k - p$ smallest elements of $L[1\,..\,n_1 + 1]$ and $R[1\,..\,n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, 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: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p\,..\,k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

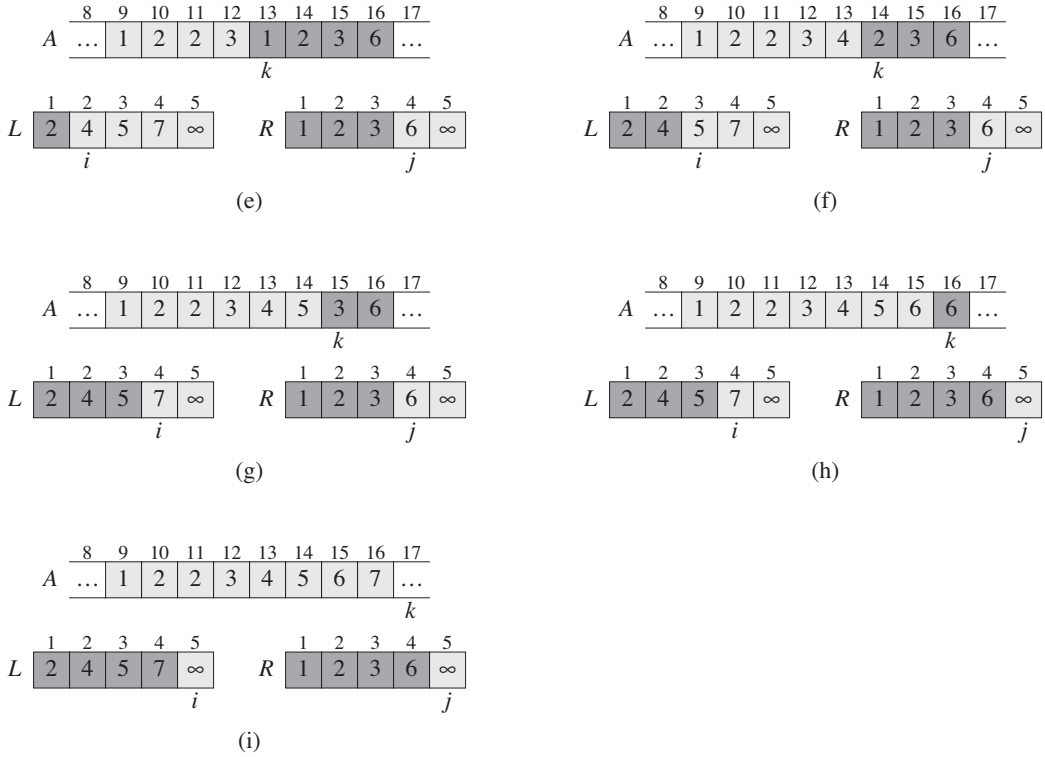


Figure 2.3, continued **(i)** The arrays and indices at termination. At this point, the subarray in $A[9\,..\,16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p\,..\,k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p\,..\,k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r+1$. By the loop invariant, the subarray $A[p\,..\,k-1]$, which is $A[p\,..\,r]$, contains the $k-p = r-p+1$ smallest elements of $L[1\,..\,n_1+1]$ and $R[1\,..\,n_2+1]$, in sorted order. The arrays L and R together contain $n_1+n_2+2 = r-p+3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of lines 1–3 and 8–11 takes constant time, the **for** loops of lines 4–7 take $\Theta(n_1 + n_2) = \Theta(n)$ time,[7] and there are n iterations of the **for** loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p..r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two subarrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.[8]

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call MERGE-SORT$(A, 1, A.length)$, where once again $A.length = n$. Figure 2.4 illustrates the operation of the procedure bottom-up when n is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n.

2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

---

[7] We shall see in Chapter 3 how to formally interpret equations containing Θ-notation.

[8] The expression $\lceil x \rceil$ denotes the least integer greater than or equal to x, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. These notations are defined in Chapter 3. The easiest way to verify that setting q to $\lfloor (p + r)/2 \rfloor$ yields subarrays $A[p..q]$ and $A[q+1..r]$ of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, is to examine the four cases that arise depending on whether each of p and r is odd or even.

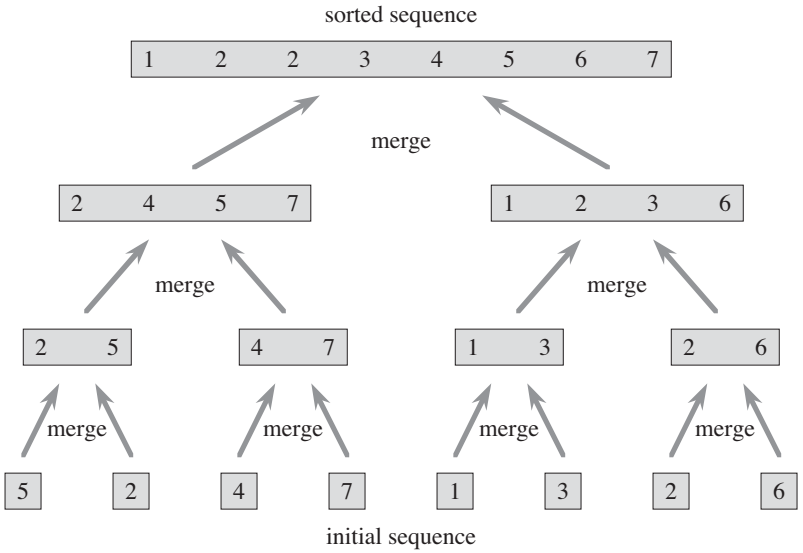


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) It takes time $T(n/b)$ to solve one subproblem of size n/b, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \ . \end{cases}$$

In Chapter 4, we shall see how to solve common recurrences of this form.

Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that

the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. In Chapter 4, we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{2.1}$$

In Chapter 4, we shall see the "master theorem," which we can use to show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

We do not need the master theorem to intuitively understand why the solution to the recurrence (2.1) is $T(n) = \Theta(n \lg n)$. Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases} \tag{2.2}$$

where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.[9]

[9] It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting c be the larger of these times and understanding that our recurrence gives an upper bound on the running time, or by letting c be the lesser of these times and understanding that our recurrence gives a lower bound on the running time. Both bounds are on the order of $n \lg n$ and, taken together, give a $\Theta(n \lg n)$ running time.

Figure 2.5 shows how we can solve recurrence (2.2). For convenience, we assume that n is an exact power of 2. Part (a) of the figure shows $T(n)$, which we expand in part (b) into an equivalent tree representing the recurrence. The cn term is the root (the cost incurred at the top level of recursion), and the two subtrees of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost incurred at each of the two subnodes at the second level of recursion is $cn/2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c. Part (d) shows the resulting ***recursion tree***.

Next, we add the costs across each level of the tree. The top level has total cost cn, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after that has total cost $c(n/4)+c(n/4)+c(n/4)+c(n/4) = cn$, and so on. In general, the level i below the top has 2^i nodes, each contributing a cost of $c(n/2^i)$, so that the ith level below the top has total cost $2^i\, c(n/2^i) = cn$. The bottom level has n nodes, each contributing a cost of c, for a total cost of cn.

The total number of levels of the recursion tree in Figure 2.5 is $\lg n + 1$, where n is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim. The base case occurs when $n = 1$, in which case the tree has only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree with 2^i leaves is $\lg 2^i + 1 = i + 1$ (since for any value of i, we have that $\lg 2^i = i$). Because we are assuming that the input size is a power of 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has one more level than a tree with 2^i leaves, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels, each costing cn, for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant c gives the desired result of $\Theta(n \lg n)$.

Exercises

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49\rangle$.

2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

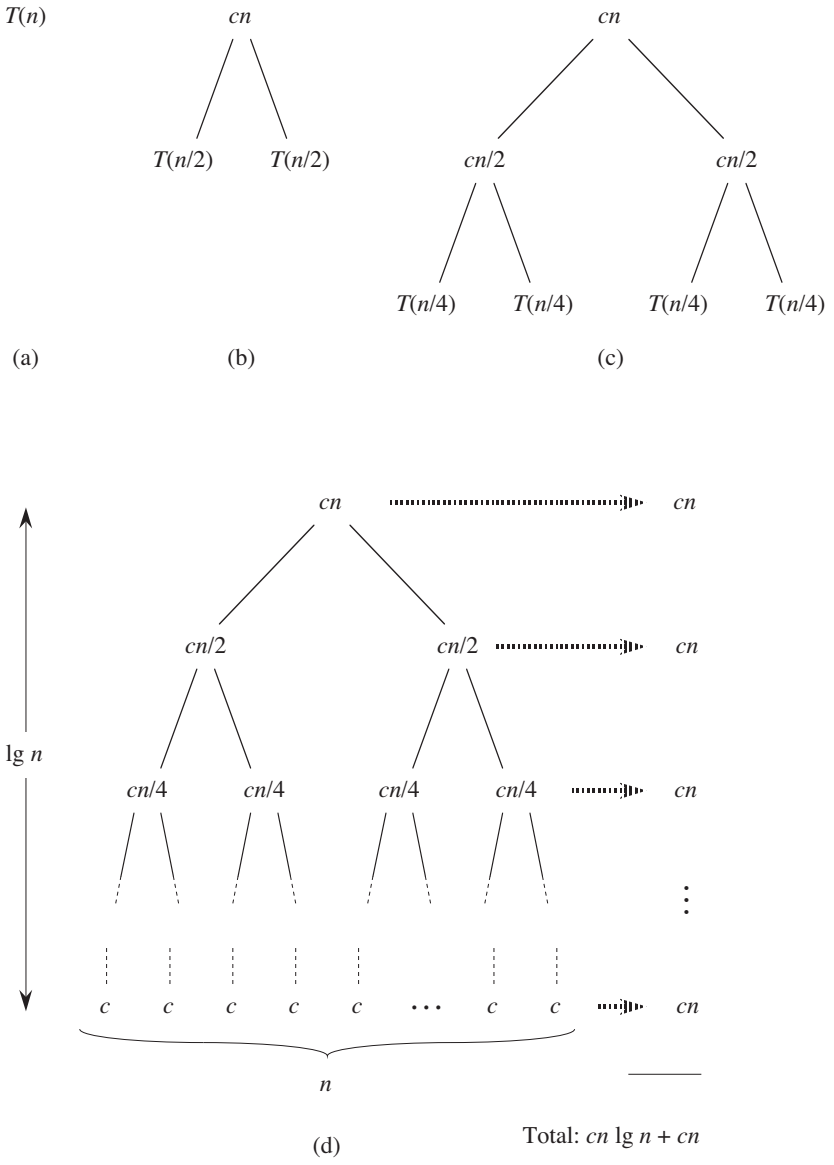


Figure 2.5 How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

2.3-3
Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2\,, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

2.3-4
We can express insertion sort as a recursive procedure as follows. In order to sort $A[1\,..\,n]$, we recursively sort $A[1\,..\,n-1]$ and then insert $A[n]$ into the sorted array $A[1\,..\,n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

2.3-5
Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

2.3-6
Observe that the **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1\,..\,j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

2.3-7 ⋆
Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Problems

2-1 Insertion sort on small arrays in merge sort
Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when

subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should we choose k in practice?

2-2 Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

a. Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n] \,, \tag{2.3}$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

2-3 Correctness of Horner's rule

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots)) \,, \end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value for x:

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

a. In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

c. Consider the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3,
>
> $$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \,.$$
>
> Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

2-4 Inversions

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.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

Chapter notes

In 1968, Knuth published the first of three volumes with the general title *The Art of Computer Programming* [209, 210, 211]. The first volume ushered in the modern study of computer algorithms with a focus on the analysis of running time, and the full series remains an engaging and worthwhile reference for many of the topics presented here. According to Knuth, the word "algorithm" is derived from the name "al-Khowârizmî," a ninth-century Persian mathematician.

Aho, Hopcroft, and Ullman [5] advocated the asymptotic analysis of algorithms—using notations that Chapter 3 introduces, including Θ-notation—as a means of comparing relative performance. They also popularized the use of recurrence relations to describe the running times of recursive algorithms.

Knuth [211] provides an encyclopedic treatment of many sorting algorithms. His comparison of sorting algorithms (page 381) includes exact step-counting analyses, like the one we performed here for insertion sort. Knuth's discussion of insertion sort encompasses several variations of the algorithm. The most important of these is Shell's sort, introduced by D. L. Shell, which uses insertion sort on periodic subsequences of the input to produce a faster sorting algorithm.

Merge sort is also described by Knuth. He mentions that a mechanical collator capable of merging two decks of punched cards in a single pass was invented in 1938. J. von Neumann, one of the pioneers of computer science, apparently wrote a program for merge sort on the EDVAC computer in 1945.

The early history of proving programs correct is described by Gries [153], who credits P. Naur with the first article in this field. Gries attributes loop invariants to R. W. Floyd. The textbook by Mitchell [256] describes more recent progress in proving programs correct.

3 Growth of Functions

The order of growth of the running time of an algorithm, defined in Chapter 2, gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. Once the input size n becomes large enough, merge sort, with its $\Theta(n \lg n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\Theta(n^2)$. Although we can sometimes determine the exact running time of an algorithm, as we did for insertion sort in Chapter 2, the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

This chapter gives several standard methods for simplifying the asymptotic analysis of algorithms. The next section begins by defining several types of "asymptotic notation," of which we have already seen an example in Θ-notation. We then present several notational conventions used throughout this book, and finally we review the behavior of functions that commonly arise in the analysis of algorithms.

3.1 Asymptotic notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$. Such notations are convenient for describing the worst-case running-time function $T(n)$, which usually is defined only on integer input sizes. We sometimes find it convenient, however, to *abuse* asymptotic notation in a va-

riety of ways. For example, we might extend the notation to the domain of real numbers or, alternatively, restrict it to a subset of the natural numbers. We should make sure, however, to understand the precise meaning of the notation so that when we abuse, we do not *misuse* it. This section defines the basic asymptotic notations and also introduces some common abuses.

Asymptotic notation, functions, and running times

We will use asymptotic notation primarily to describe the running times of algorithms, as when we wrote that insertion sort's worst-case running time is $\Theta(n^2)$. Asymptotic notation actually applies to functions, however. Recall that we characterized insertion sort's worst-case running time as an^2+bn+c, for some constants a, b, and c. By writing that insertion sort's running time is $\Theta(n^2)$, we abstracted away some details of this function. Because asymptotic notation applies to functions, what we were writing as $\Theta(n^2)$ was the function $an^2 + bn + c$, which in that case happened to characterize the worst-case running time of insertion sort.

In this book, the functions to which we apply asymptotic notation will usually characterize the running times of algorithms. But asymptotic notation can apply to functions that characterize some other aspect of algorithms (the amount of space they use, for example), or even to functions that have nothing whatsoever to do with algorithms.

Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand *which* running time we mean. Sometimes we are interested in the worst-case running time. Often, however, we wish to characterize the running time no matter what the input. In other words, we often wish to make a blanket statement that covers all inputs, not just the worst case. We shall see asymptotic notations that are well suited to characterizing running times no matter what the input.

Θ-notation

In Chapter 2, we found that the worst-case running time of insertion sort is $T(n) = \Theta(n^2)$. Let us define what this notation means. For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$[1]

[1] Within set notation, a colon means "such that."

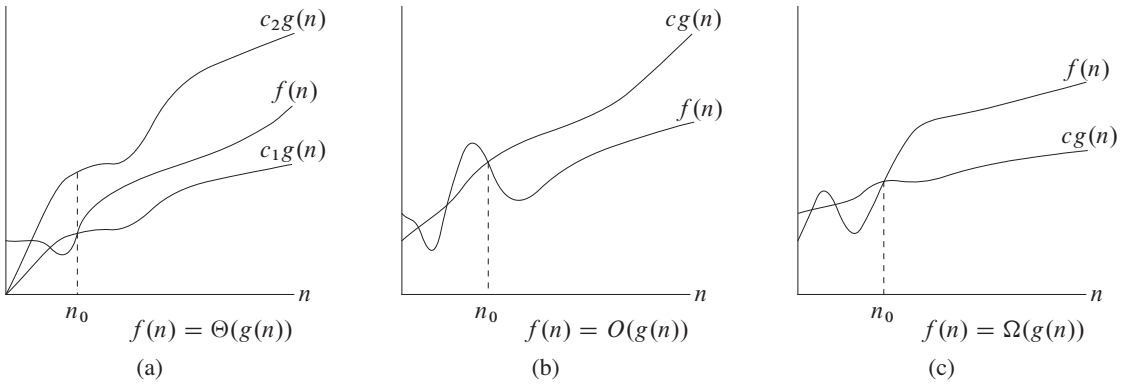


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$. Instead, we will usually write "$f(n) = \Theta(g(n))$" to express the same notion. You might be confused because we abuse equality in this way, but we shall see later in this section that doing so has its advantages.

Figure 3.1(a) gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n) = \Theta(g(n))$. For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be ***asymptotically nonnegative***, that is, that $f(n)$ be nonnegative whenever n is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within Θ-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

In Chapter 2, we introduced an informal notion of Θ-notation that amounted to throwing away lower-order terms and ignoring the leading coefficient of the highest-order term. Let us briefly justify this intuition by using the formal definition to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants c_1, c_2, and n_0 such that

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

for all $n \geq n_0$. Dividing by n^2 yields

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

We can make the right-hand inequality hold for any value of $n \geq 1$ by choosing any constant $c_2 \geq 1/2$. Likewise, we can make the left-hand inequality hold for any value of $n \geq 7$ by choosing any constant $c_1 \leq 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that *some* choice exists. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants.

We can also use the formal definition to verify that $6n^3 \neq \Theta(n^2)$. Suppose for the purpose of contradiction that c_2 and n_0 exist such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$. But then dividing by n^2 yields $n \leq c_2/6$, which cannot possibly hold for arbitrarily large n, since c_2 is constant.

Intuitively, the lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n. When n is large, even a tiny fraction of the highest-order term suffices to dominate the lower-order terms. Thus, setting c_1 to a value that is slightly smaller than the coefficient of the highest-order term and setting c_2 to a value that is slightly larger permits the inequalities in the definition of Θ-notation to be satisfied. The coefficient of the highest-order term can likewise be ignored, since it only changes c_1 and c_2 by a constant factor equal to the coefficient.

As an example, consider any quadratic function $f(n) = an^2 + bn + c$, where a, b, and c are constants and $a > 0$. Throwing away the lower-order terms and ignoring the constant yields $f(n) = \Theta(n^2)$. Formally, to show the same thing, we take the constants $c_1 = a/4$, $c_2 = 7a/4$, and $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$. You may verify that $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ for all $n \geq n_0$. In general, for any polynomial $p(n) = \sum_{i=0}^{d} a_i n^i$, where the a_i are constants and $a_d > 0$, we have $p(n) = \Theta(n^d)$ (see Problem 3-1).

Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$, or $\Theta(1)$. This latter notation is a minor abuse, however, because the

expression does not indicate what variable is tending to infinity.[2] We shall often use the notation $\Theta(1)$ to mean either a constant or a constant function with respect to some variable.

O-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

We use O-notation to give an upper bound on a function, to within a constant factor. Figure 3.1(b) shows the intuition behind O-notation. For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $cg(n)$.

We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than O-notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$. Thus, our proof that any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\Theta(n^2)$ also shows that any such quadratic function is in $O(n^2)$. What may be more surprising is that when $a > 0$, any *linear* function $an + b$ is in $O(n^2)$, which is easily verified by taking $c = a + |b|$ and $n_0 = \max(1, -b/a)$.

If you have seen O-notation before, you might find it strange that we should write, for example, $n = O(n^2)$. In the literature, we sometimes find O-notation informally describing asymptotically tight bounds, that is, what we have defined using Θ-notation. In this book, however, when we write $f(n) = O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is. Distinguishing asymptotic upper bounds from asymptotically tight bounds is standard in the algorithms literature.

Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm from Chapter 2 immediately yields an $O(n^2)$ upper bound on the worst-case running time: the cost of each iteration of the inner loop is bounded from above by $O(1)$ (constant), the indices i

---

[2] The real problem is that our ordinary notation for functions does not distinguish functions from values. In λ-calculus, the parameters to a function are clearly specified: the function n^2 could be written as $\lambda n.n^2$, or even $\lambda r.r^2$. Adopting a more rigorous notation, however, would complicate algebraic manipulations, and so we choose to tolerate the abuse.

and j are both at most n, and the inner loop is executed at most once for each of the n^2 pairs of values for i and j.

Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input—the blanket statement we discussed earlier. Thus, the $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input. The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\Theta(n^2)$ bound on the running time of insertion sort on *every* input. For example, we saw in Chapter 2 that when the input is already sorted, insertion sort runs in $\Theta(n)$ time.

Technically, it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given n, the actual running time varies, depending on the particular input of size n. When we say "the running time is $O(n^2)$," we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$. Equivalently, we mean that the worst-case running time is $O(n^2)$.

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

Figure 3.1(c) shows the intuition behind Ω-notation. For all values n at or to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

From the definitions of the asymptotic notations we have seen thus far, it is easy to prove the following important theorem (see Exercise 3.1-5).

Theorem 3.1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

As an example of the application of this theorem, our proof that $an^2 + bn + c = \Theta(n^2)$ for any constants a, b, and c, where $a > 0$, immediately implies that $an^2 + bn + c = \Omega(n^2)$ and $an^2 + bn + c = O(n^2)$. In practice, rather than using Theorem 3.1 to obtain asymptotic upper and lower bounds from asymptotically tight bounds, as we did for this example, we usually use it to prove asymptotically tight bounds from asymptotic upper and lower bounds.

When we say that the *running time* (no modifier) of an algorithm is $\Omega(g(n))$, we mean that *no matter what particular input of size n is chosen for each value of n*, the running time on that input is at least a constant times $g(n)$, for sufficiently large n. Equivalently, we are giving a lower bound on the best-case running time of an algorithm. For example, the best-case running time of insertion sort is $\Omega(n)$, which implies that the running time of insertion sort is $\Omega(n)$.

The running time of insertion sort therefore belongs to both $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of n and a quadratic function of n. Moreover, these bounds are asymptotically as tight as possible: for instance, the running time of insertion sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted). It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Asymptotic notation in equations and inequalities

We have already seen how asymptotic notation can be used within mathematical formulas. For example, in introducing O-notation, we wrote "$n = O(n^2)$." We might also write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. How do we interpret such formulas?

When the asymptotic notation stands alone (that is, not within a larger formula) on the right-hand side of an equation (or inequality), as in $n = O(n^2)$, we have already defined the equal sign to mean set membership: $n \in O(n^2)$. In general, however, when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n)$ is some function in the set $\Theta(n)$. In this case, we let $f(n) = 3n + 1$, which indeed is in $\Theta(n)$.

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, in Chapter 2 we expressed the worst-case running time of merge sort as the recurrence

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

If we are interested only in the asymptotic behavior of $T(n)$, there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. For example, in the expression

$$\sum_{i=1}^{n} O(i) ,$$

there is only a single anonymous function (a function of i). This expression is thus *not* the same as $O(1) + O(2) + \cdots + O(n)$, which doesn't really have a clean interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation, as in

$$2n^2 + \Theta(n) = \Theta(n^2) .$$

We interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.* Thus, our example means that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

We can chain together a number of such relationships, as in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

We can interpret each equation separately by the rules above. The first equation says that there is *some* function $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$ for all n. The second equation says that for *any* function $g(n) \in \Theta(n)$ (such as the $f(n)$ just mentioned), there is *some* function $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$ for all n. Note that this interpretation implies that $2n^2 + 3n + 1 = \Theta(n^2)$, which is what the chaining of equations intuitively gives us.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$$\begin{aligned} o(g(n)) = \{ f(n) : &\text{ for any positive constant } c > 0, \text{ there exists a constant} \\ &n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \} . \end{aligned}$$

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \,. \tag{3.1}$$

Some authors use this limit as a definition of the o-notation; the definition in this book also restricts the anonymous functions to be asymptotically nonnegative.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} \,.$$

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

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

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

Comparing functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that $f(n)$ and $g(n)$ are asymptotically positive.

Transitivity:

$$\begin{aligned}
&f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) \,,\\
&f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) \,,\\
&f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) \,,\\
&f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) \,,\\
&f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)) \,.
\end{aligned}$$

Reflexivity:

$$\begin{aligned}
f(n) &= \Theta(f(n)) \,,\\
f(n) &= O(f(n)) \,,\\
f(n) &= \Omega(f(n)) \,.
\end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) ,$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .$$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$f(n) = O(g(n)) \quad \text{is like} \quad a \leq b ,$$
$$f(n) = \Omega(g(n)) \quad \text{is like} \quad a \geq b ,$$
$$f(n) = \Theta(g(n)) \quad \text{is like} \quad a = b ,$$
$$f(n) = o(g(n)) \quad \text{is like} \quad a < b ,$$
$$f(n) = \omega(g(n)) \quad \text{is like} \quad a > b .$$

We say that $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

One property of real numbers, however, does not carry over to asymptotic notation:

Trichotomy: For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, we cannot compare the functions n and $n^{1+\sin n}$ using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

Exercises

3.1-1

Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1-2

Show that for any real constants a and b, where $b > 0$,

$$(n + a)^b = \Theta(n^b) . \tag{3.2}$$

3.1-3
Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

3.1-4
Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

3.1-5
Prove Theorem 3.1.

3.1-6
Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

3.1-7
Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

3.1-8
We can extend our notation to the case of two parameters n and m that can go to infinity independently at different rates. For a given function $g(n,m)$, we denote by $O(g(n,m))$ the set of functions

$$O(g(n,m)) = \{f(n,m) : \text{ there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \le f(n,m) \le cg(n,m) \text{ for all } n \ge n_0 \text{ or } m \ge m_0\} .$$

Give corresponding definitions for $\Omega(g(n,m))$ and $\Theta(g(n,m))$.

3.2 Standard notations and common functions

This section reviews some standard mathematical functions and notations and explores the relationships among them. It also illustrates the use of the asymptotic notations.

Monotonicity

A function $f(n)$ is ***monotonically increasing*** if $m \le n$ implies $f(m) \le f(n)$. Similarly, it is ***monotonically decreasing*** if $m \le n$ implies $f(m) \ge f(n)$. A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). For all real x,

$$x - 1 \ < \ \lfloor x \rfloor \ \leq \ x \ \leq \ \lceil x \rceil \ < \ x + 1 \ . \tag{3.3}$$

For any integer n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n \ ,$$

and for any real number $x \geq 0$ and integers $a, b > 0$,

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil \ = \ \left\lceil \frac{x}{ab} \right\rceil \ , \tag{3.4}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor \ = \ \left\lfloor \frac{x}{ab} \right\rfloor \ , \tag{3.5}$$

$$\left\lceil \frac{a}{b} \right\rceil \ \leq \ \frac{a + (b-1)}{b} \ , \tag{3.6}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \ \geq \ \frac{a - (b-1)}{b} \ . \tag{3.7}$$

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

Modular arithmetic

For any integer a and any positive integer n, the value $a \bmod n$ is the ***remainder*** (or ***residue***) of the quotient a/n:

$$a \bmod n = a - n \lfloor a/n \rfloor \ . \tag{3.8}$$

It follows that

$$0 \leq a \bmod n < n \ . \tag{3.9}$$

Given a well-defined notion of the remainder of one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(a \bmod n) = (b \bmod n)$, we write $a \equiv b \pmod{n}$ and say that a is ***equivalent*** to b, modulo n. In other words, $a \equiv b \pmod{n}$ if a and b have the same remainder when divided by n. Equivalently, $a \equiv b \pmod{n}$ if and only if n is a divisor of $b - a$. We write $a \not\equiv b \pmod{n}$ if a is not equivalent to b, modulo n.

Polynomials

Given a nonnegative integer d, a ***polynomial in n of degree d*** is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where the constants $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$. For an asymptotically positive polynomial $p(n)$ of degree d, we have $p(n) = \Theta(n^d)$. For any real constant $a \geq 0$, the function n^a is monotonically increasing, and for any real constant $a \leq 0$, the function n^a is monotonically decreasing. We say that a function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant k.

Exponentials

For all real $a > 0$, m, and n, we have the following identities:

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

For all n and $a \geq 1$, the function a^n is monotonically increasing in n. When convenient, we shall assume $0^0 = 1$.

We can relate the rates of growth of polynomials and exponentials by the following fact. For all real constants a and b such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 , \tag{3.10}$$

from which we can conclude that

$$n^b = o(a^n) .$$

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

Using e to denote 2.71828..., the base of the natural logarithm function, we have for all real x,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} , \tag{3.11}$$

where "!" denotes the factorial function defined later in this section. For all real x, we have the inequality

$$e^x \geq 1 + x \,, \tag{3.12}$$

where equality holds only when $x = 0$. When $|x| \leq 1$, we have the approximation

$$1 + x \leq e^x \leq 1 + x + x^2 \,. \tag{3.13}$$

When $x \to 0$, the approximation of e^x by $1 + x$ is quite good:

$$e^x = 1 + x + \Theta(x^2) \,.$$

(In this equation, the asymptotic notation is used to describe the limiting behavior as $x \to 0$ rather than as $x \to \infty$.) We have for all x,

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x \,. \tag{3.14}$$

Logarithms

We shall use the following notations:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} \,, \\
\ln n &= \log_e n && \text{(natural logarithm)} \,, \\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} \,, \\
\lg \lg n &= \lg(\lg n) && \text{(composition)} \,.
\end{aligned}$$

An important notational convention we shall adopt is that *logarithm functions will apply only to the next term in the formula*, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n + k)$. If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$, and n,

$$\begin{aligned}
a &= b^{\log_b a} \,, \\
\log_c(ab) &= \log_c a + \log_c b \,, \\
\log_b a^n &= n \log_b a \,, \\
\log_b a &= \frac{\log_c a}{\log_c b} \,, && (3.15) \\
\log_b(1/a) &= -\log_b a \,, \\
\log_b a &= \frac{1}{\log_a b} \,, \\
a^{\log_b c} &= c^{\log_b a} \,, && (3.16)
\end{aligned}$$

where, in each equation above, logarithm bases are not 1.

By equation (3.15), changing the base of a logarithm from one constant to another changes the value of the logarithm by only a constant factor, and so we shall often use the notation "$\lg n$" when we don't care about constant factors, such as in O-notation. Computer scientists find 2 to be the most natural base for logarithms because so many algorithms and data structures involve splitting a problem into two parts.

There is a simple series expansion for $\ln(1+x)$ when $|x| < 1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots .$$

We also have the following inequalities for $x > -1$:

$$\frac{x}{1+x} \le \ln(1+x) \le x , \tag{3.17}$$

where equality holds only for $x = 0$.

We say that a function $f(n)$ is ***polylogarithmically bounded*** if $f(n) = O(\lg^k n)$ for some constant k. We can relate the growth of polynomials and polylogarithms by substituting $\lg n$ for n and 2^a for a in equation (3.10), yielding

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 .$$

From this limit, we can conclude that

$$\lg^b n = o(n^a)$$

for any constant $a > 0$. Thus, any positive polynomial function grows faster than any polylogarithmic function.

Factorials

The notation $n!$ (read "n factorial") is defined for integers $n \ge 0$ as

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

Thus, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

A weak upper bound on the factorial function is $n! \le n^n$, since each of the n terms in the factorial product is at most n. ***Stirling's approximation***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \tag{3.18}$$

where e is the base of the natural logarithm, gives us a tighter upper bound, and a lower bound as well. As Exercise 3.2-3 asks you to prove,

$$\begin{aligned} n! &= o(n^n) , \\ n! &= \omega(2^n) , \\ \lg(n!) &= \Theta(n \lg n) , \end{aligned} \tag{3.19}$$

where Stirling's approximation is helpful in proving equation (3.19). The following equation also holds for all $n \geq 1$:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.20}$$

where

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} . \tag{3.21}$$

Functional iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied i times to an initial value of n. Formally, let $f(n)$ be a function over the reals. For nonnegative integers i, we recursively define

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases}$$

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

The iterated logarithm function

We use the notation $\lg^* n$ (read "log star of n") to denote the iterated logarithm, defined as follows. Let $\lg^{(i)} n$ be as defined above, with $f(n) = \lg n$. Because the logarithm of a nonpositive number is undefined, $\lg^{(i)} n$ is defined only if $\lg^{(i-1)} n > 0$. Be sure to distinguish $\lg^{(i)} n$ (the logarithm function applied i times in succession, starting with argument n) from $\lg^i n$ (the logarithm of n raised to the ith power). Then we define the iterated logarithm function as

$$\lg^* n = \min \left\{ i \geq 0 : \lg^{(i)} n \leq 1 \right\} .$$

The iterated logarithm is a *very* slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1 , \\ \lg^* 4 &= 2 , \\ \lg^* 16 &= 3 , \\ \lg^* 65536 &= 4 , \\ \lg^*(2^{65536}) &= 5 . \end{aligned}$$

Since the number of atoms in the observable universe is estimated to be about 10^{80}, which is much less than 2^{65536}, we rarely encounter an input size n such that $\lg^* n > 5$.

Fibonacci numbers

We define the ***Fibonacci numbers*** by the following recurrence:

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 . \end{aligned} \tag{3.22}$$

Thus, each Fibonacci number is the sum of the two previous ones, yielding the sequence

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots\ .$$

Fibonacci numbers are related to the ***golden ratio*** ϕ and to its conjugate $\widehat{\phi}$, which are the two roots of the equation

$$x^2 = x + 1 \tag{3.23}$$

and are given by the following formulas (see Exercise 3.2-6):

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots , \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots . \end{aligned} \tag{3.24}$$

Specifically, we have

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} ,$$

which we can prove by induction (Exercise 3.2-7). Since $|\widehat{\phi}| < 1$, we have

$$\begin{aligned} \frac{|\widehat{\phi}^i|}{\sqrt{5}} &< \frac{1}{\sqrt{5}} \\ &< \frac{1}{2} , \end{aligned}$$

which implies that

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor , \tag{3.25}$$

which is to say that the ith Fibonacci number F_i is equal to $\phi^i / \sqrt{5}$ rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially.

Exercises

3.2-1
Show that if $f(n)$ and $g(n)$ are monotonically increasing functions, then so are the functions $f(n) + g(n)$ and $f(g(n))$, and if $f(n)$ and $g(n)$ are in addition nonnegative, then $f(n) \cdot g(n)$ is monotonically increasing.

3.2-2
Prove equation (3.16).

3.2-3
Prove equation (3.19). Also prove that $n! = \omega(2^n)$ and $n! = o(n^n)$.

3.2-4 ⋆
Is the function $\lceil \lg n \rceil !$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil !$ polynomially bounded?

3.2-5 ⋆
Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

3.2-6
Show that the golden ratio ϕ and its conjugate $\widehat{\phi}$ both satisfy the equation $x^2 = x + 1$.

3.2-7
Prove by induction that the ith Fibonacci number satisfies the equality

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} ,$$

where ϕ is the golden ratio and $\widehat{\phi}$ is its conjugate.

3.2-8
Show that $k \ln k = \Theta(n)$ implies $k = \Theta(n / \ln n)$.

Problems

3-1 Asymptotic behavior of polynomials

Let

$$p(n) = \sum_{i=0}^{d} a_i n^i \ ,$$

where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.

a. If $k \geq d$, then $p(n) = O(n^k)$.

b. If $k \leq d$, then $p(n) = \Omega(n^k)$.

c. If $k = d$, then $p(n) = \Theta(n^k)$.

d. If $k > d$, then $p(n) = o(n^k)$.

e. If $k < d$, then $p(n) = \omega(n^k)$.

3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below, whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-3 Ordering by asymptotic growth rates

a. Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

3-4 Asymptotic notation properties

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O\left(2^{g(n)}\right)$.

e. $f(n) = O\left((f(n))^2\right)$.

f. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

3-5 Variations on O and Ω

Some authors define Ω in a slightly different way than we do; let's use $\overset{\infty}{\Omega}$ (read "omega infinity") for this alternative definition. We say that $f(n) = \overset{\infty}{\Omega}(g(n))$ if there exists a positive constant c such that $f(n) \geq cg(n) \geq 0$ for infinitely many integers n.

a. Show that for any two functions $f(n)$ and $g(n)$ that are asymptotically nonnegative, either $f(n) = O(g(n))$ or $f(n) = \overset{\infty}{\Omega}(g(n))$ or both, whereas this is not true if we use Ω in place of $\overset{\infty}{\Omega}$.

b. Describe the potential advantages and disadvantages of using $\overset{\infty}{\Omega}$ instead of Ω to characterize the running times of programs.

Some authors also define O in a slightly different manner; let's use O' for the alternative definition. We say that $f(n) = O'(g(n))$ if and only if $|f(n)| = O(g(n))$.

c. What happens to each direction of the "if and only if" in Theorem 3.1 if we substitute O' for O but still use Ω?

Some authors define $\widetilde{O}$ (read "soft-oh") to mean O with logarithmic factors ignored:

$$\widetilde{O}(g(n)) = \{f(n) : \text{ there exist positive constants } c, k, \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n)\lg^k(n) \text{ for all } n \ge n_0\} .$$

d. Define $\widetilde{\Omega}$ and $\widetilde{\Theta}$ in a similar manner. Prove the corresponding analog to Theorem 3.1.

3-6 Iterated functions

We can apply the iteration operator * used in the $\lg^*$ function to any monotonically increasing function $f(n)$ over the reals. For a given constant $c \in \mathbb{R}$, we define the iterated function f_c^* by

$$f_c^*(n) = \min\left\{i \ge 0 : f^{(i)}(n) \le c\right\} ,$$

which need not be well defined in all cases. In other words, the quantity $f_c^*(n)$ is the number of iterated applications of the function f required to reduce its argument down to c or less.

For each of the following functions $f(n)$ and constants c, give as tight a bound as possible on $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

Chapter notes

Knuth [209] traces the origin of the O-notation to a number-theory text by P. Bachmann in 1892. The o-notation was invented by E. Landau in 1909 for his discussion of the distribution of prime numbers. The Ω and Θ notations were advocated by Knuth [213] to correct the popular, but technically sloppy, practice in the literature of using O-notation for both upper and lower bounds. Many people continue to use the O-notation where the Θ-notation is more technically precise. Further discussion of the history and development of asymptotic notations appears in works by Knuth [209, 213] and Brassard and Bratley [54].

Not all authors define the asymptotic notations in the same way, although the various definitions agree in most common situations. Some of the alternative definitions encompass functions that are not asymptotically nonnegative, as long as their absolute values are appropriately bounded.

Equation (3.20) is due to Robbins [297]. Other properties of elementary mathematical functions can be found in any good mathematical reference, such as Abramowitz and Stegun [1] or Zwillinger [362], or in a calculus book, such as Apostol [18] or Thomas et al. [334]. Knuth [209] and Graham, Knuth, and Patashnik [152] contain a wealth of material on discrete mathematics as used in computer science.

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

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

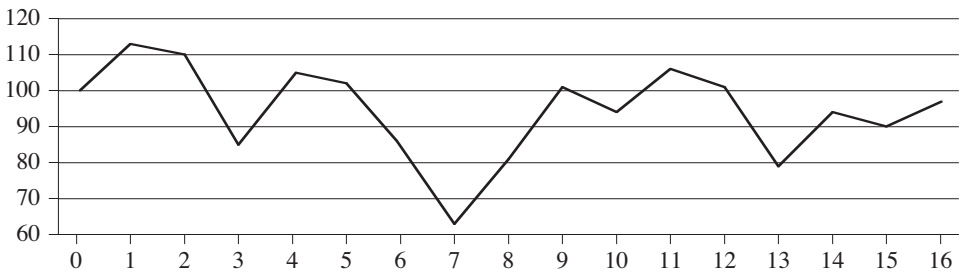
without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

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

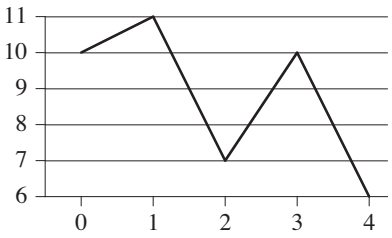
Suppose that you been offered the opportunity to invest in the Volatile Chemical Corporation. Like the chemicals the company produces, the stock price of the Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit of stock only one time and then sell it at a later date, buying 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

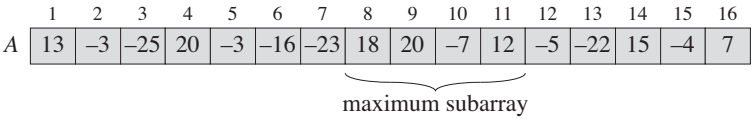


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8\,..\,11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

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

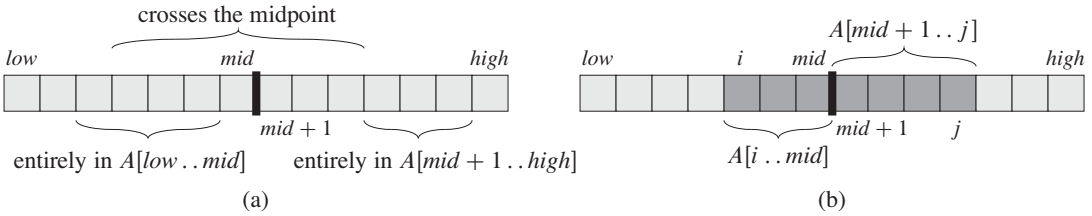


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ 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 *left-sum* + *right-sum* of the values in the subarray $A[max\text{-}left\,..\,max\text{-}right]$.

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 $A[1..n]$.

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 $A[low..mid]$ as the ***left subarray*** and to $A[mid + 1..high]$ as the ***right subarray***. Because we know that the subarray $A[low..high]$ 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 $A[1..j]$, extend the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray of $A[1..j+1]$ is either a maximum subarray of $A[1..j]$ or a subarray $A[i..j+1]$, for some $1 \le i \le j+1$. Determine a maximum subarray of the form $A[i..j+1]$ 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 × 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 · 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 ma-

trices 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

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

which corresponds to equation (4.11).

Similarly, we set

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

and so C_{12} equals

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

corresponding to equation (4.12).

Setting

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

makes C_{21} equal

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

corresponding to equation (4.13).

Finally, we set

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

so that C_{22} equals

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

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 $T(n) \leq cn^{\log_3 4}$ 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 $T(n) \leq cn^2$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

4.3-9

Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ 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.

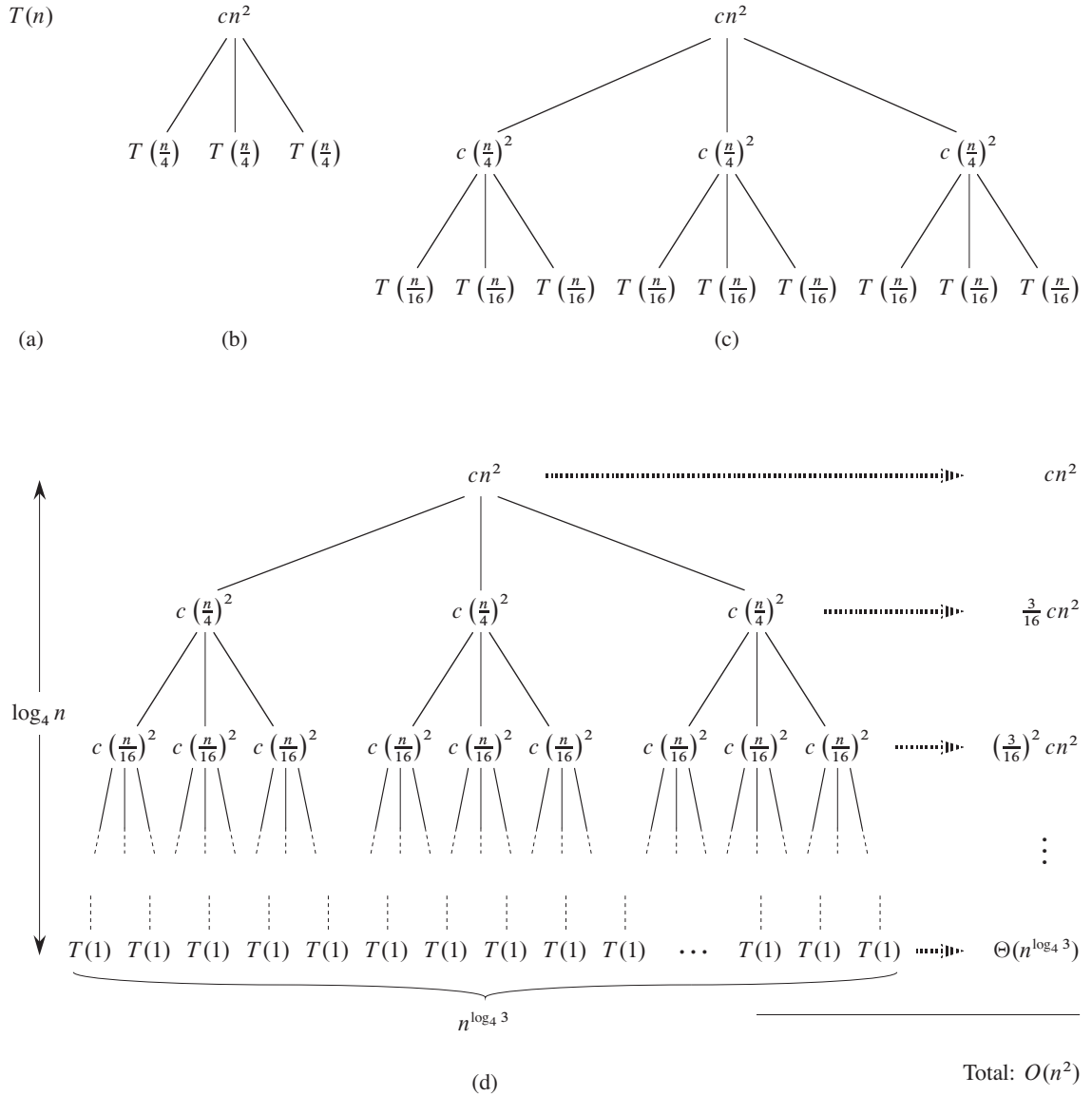


Figure 4.5 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

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

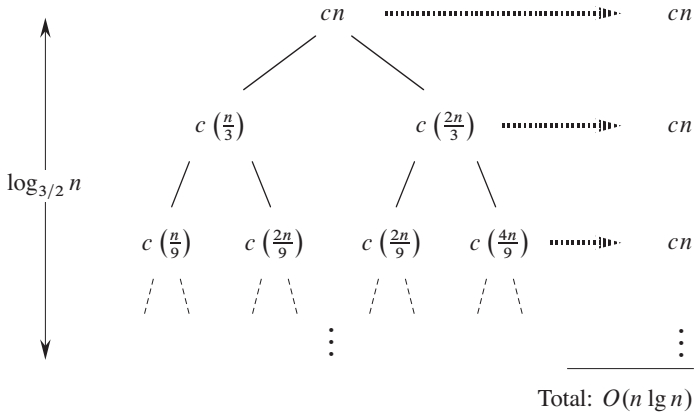


Figure 4.6 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

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

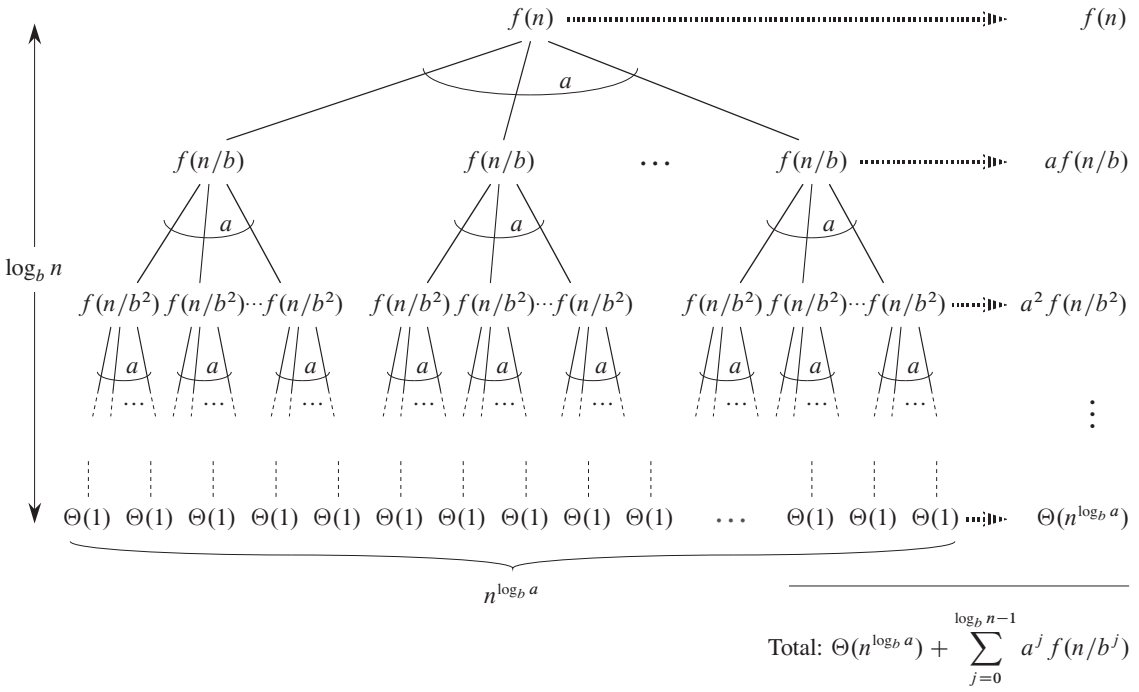


Figure 4.7 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of the nodes at each depth is shown at the right, and their sum is given in equation (4.21).

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:

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

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

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

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

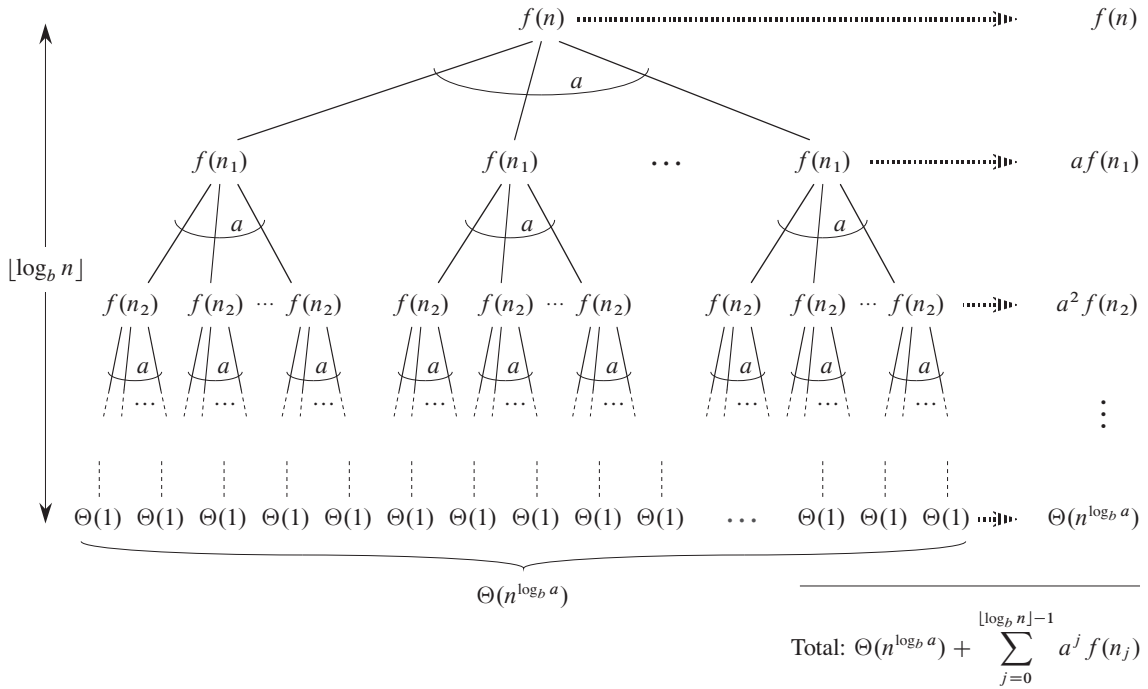


Figure 4.8 The recursion tree generated by $T(n) = aT(\lceil n/b \rceil) + f(n)$. The recursive argument n_j is given by equation (4.27).

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

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

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 $n \leq 2$. 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 $i = 1, 2, ..., m - 1$ and $j = 1, 2, ..., n - 1$, 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 $f(1) \leq f(2) \leq \cdots \leq f(m)$ 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.

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 $\mathrm{E}[X]$:

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

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 $A[1\,..\,n]$ 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 $A[i\,..\,n]$, 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[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 = 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 ∅
3  else S = RANDOM-SAMPLE(m − 1, n − 1)
4      i = RANDOM(1, n)
5      if i ∈ S
6          S = S ∪ {n}
7      else S = S ∪ {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

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

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

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} , \tag{5.7}$$

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].

II Sorting and Order Statistics

Introduction

This part presents several algorithms that solve the following ***sorting problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The input sequence is usually an n-element array, although it may be represented in some other fashion, such as a linked list.

The structure of the data

In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a ***record***. Each record contains a ***key***, which is the value to be sorted. The remainder of the record consists of ***satellite data***, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

In a sense, it is these implementation details that distinguish an algorithm from a full-blown program. A sorting algorithm describes the *method* by which we determine the sorted order, regardless of whether we are sorting individual numbers or large records containing many bytes of satellite data. Thus, when focusing on the problem of sorting, we typically assume that the input consists only of numbers. Translating an algorithm for sorting numbers into a program for sorting records

is conceptually straightforward, although in a given engineering situation other subtleties may make the actual programming task a challenge.

Why sorting?

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms. There are several reasons:

- Sometimes an application inherently needs to sort information. For example, in order to prepare customer statements, banks need to sort checks by check number.
- Algorithms often use sorting as a key subroutine. For example, a program that renders graphical objects which are layered on top of each other might have to sort the objects according to an "above" relation so that it can draw these objects from bottom to top. We shall see numerous algorithms in this text that use sorting as a subroutine.
- We can draw from among a wide variety of sorting algorithms, and they employ a rich set of techniques. In fact, many important techniques used throughout algorithm design appear in the body of sorting algorithms that have been developed over the years. In this way, sorting is also a problem of historical interest.
- We can prove a nontrivial lower bound for sorting (as we shall do in Chapter 8). Our best upper bounds match the lower bound asymptotically, and so we know that our sorting algorithms are asymptotically optimal. Moreover, we can use the lower bound for sorting to prove lower bounds for certain other problems.
- Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors, such as prior knowledge about the keys and satellite data, the memory hierarchy (caches and virtual memory) of the host computer, and the software environment. Many of these issues are best dealt with at the algorithmic level, rather than by "tweaking" the code.

Sorting algorithms

We introduced two algorithms that sort n real numbers in Chapter 2. Insertion sort takes $\Theta(n^2)$ time in the worst case. Because its inner loops are tight, however, it is a fast in-place sorting algorithm for small input sizes. (Recall that a sorting algorithm sorts ***in place*** if only a constant number of elements of the input array are ever stored outside the array.) Merge sort has a better asymptotic running time, $\Theta(n \lg n)$, but the MERGE procedure it uses does not operate in place.

In this part, we shall introduce two more algorithms that sort arbitrary real numbers. Heapsort, presented in Chapter 6, sorts n numbers in place in $O(n \lg n)$ time. It uses an important data structure, called a heap, with which we can also implement a priority queue.

Quicksort, in Chapter 7, also sorts n numbers in place, but its worst-case running time is $\Theta(n^2)$. Its expected running time is $\Theta(n \lg n)$, however, and it generally outperforms heapsort in practice. Like insertion sort, quicksort has tight code, and so the hidden constant factor in its running time is small. It is a popular algorithm for sorting large input arrays.

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements. Chapter 8 begins by introducing the decision-tree model in order to study the performance limitations of comparison sorts. Using this model, we prove a lower bound of $\Omega(n \lg n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

Chapter 8 then goes on to show that we can beat this lower bound of $\Omega(n \lg n)$ if we can gather information about the sorted order of the input by means other than comparing elements. The counting sort algorithm, for example, assumes that the input numbers are in the set $\{0, 1, \ldots, k\}$. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in $\Theta(k + n)$ time. Thus, when $k = O(n)$, counting sort runs in time that is linear in the size of the input array. A related algorithm, radix sort, can be used to extend the range of counting sort. If there are n integers to sort, each integer has d digits, and each digit can take on up to k possible values, then radix sort can sort the numbers in $\Theta(d(n + k))$ time. When d is a constant and k is $O(n)$, radix sort runs in linear time. A third algorithm, bucket sort, requires knowledge of the probabilistic distribution of numbers in the input array. It can sort n real numbers uniformly distributed in the half-open interval $[0, 1)$ in average-case $O(n)$ time.

The following table summarizes the running times of the sorting algorithms from Chapters 2 and 6–8. As usual, n denotes the number of items to sort. For counting sort, the items to sort are integers in the set $\{0, 1, \ldots, k\}$. For radix sort, each item is a d-digit number, where each digit takes on k possible values. For bucket sort, we assume that the keys are real numbers uniformly distributed in the half-open interval $[0, 1)$. The rightmost column gives the average-case or expected running time, indicating which it gives when it differs from the worst-case running time. We omit the average-case running time of heapsort because we do not analyze it in this book.

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Order statistics

The ith order statistic of a set of n numbers is the ith smallest number in the set. We can, of course, select the ith order statistic by sorting the input and indexing the ith element of the output. With no assumptions about the input distribution, this method runs in $\Omega(n \lg n)$ time, as the lower bound proved in Chapter 8 shows.

In Chapter 9, we show that we can find the ith smallest element in $O(n)$ time, even when the elements are arbitrary real numbers. We present a randomized algorithm with tight pseudocode that runs in $\Theta(n^2)$ time in the worst case, but whose expected running time is $O(n)$. We also give a more complicated algorithm that runs in $O(n)$ worst-case time.

Background

Although most of this part does not rely on difficult mathematics, some sections do require mathematical sophistication. In particular, analyses of quicksort, bucket sort, and the order-statistic algorithm use probability, which is reviewed in Appendix C, and the material on probabilistic analysis and randomized algorithms in Chapter 5. The analysis of the worst-case linear-time algorithm for order statistics involves somewhat more sophisticated mathematics than the other worst-case analyses in this part.

6 Heapsort

In this chapter, we introduce another sorting algorithm: heapsort. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a "heap," to manage information. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

The term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage," such as the programming languages Java and Lisp provide. Our heap data structure is *not* garbage-collected storage, and whenever we refer to heaps in this book, we shall mean a data structure rather than an aspect of garbage collection.

6.1 Heaps

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: $A.length$, which (as usual) gives the number of elements in the array, and $A.heap\text{-}size$, which represents how many elements in the heap are stored within array A. That is, although $A[1..A.length]$ may contain numbers, only the elements in $A[1..A.heap\text{-}size]$, where $0 \le A.heap\text{-}size \le A.length$, are valid elements of the heap. The root of the tree is $A[1]$, and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

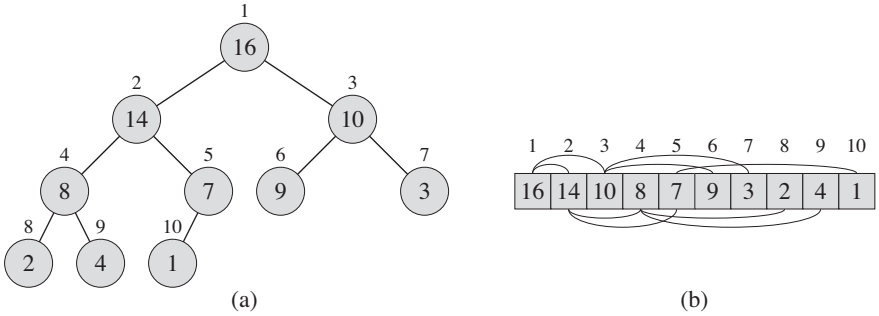


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left by one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position. Good implementations of heapsort often implement these procedures as "macros" or "in-line" procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains

values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \leq A[i] .$$

The smallest element in a min-heap is at the root.

For the heapsort algorithm, we use max-heaps. Min-heaps commonly implement priority queues, which we discuss in Section 6.5. We shall be precise in specifying whether we need a max-heap or a min-heap for any particular application, and when properties apply to either max-heaps or min-heaps, we just use the term "heap."

Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$ (see Exercise 6.1-2). We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents some basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Exercises

6.1-1

What are the minimum and maximum numbers of elements in a heap of height h?

6.1-2

Show that an n-element heap has height $\lfloor \lg n \rfloor$.

6.1-3

Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

6.1-4
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

6.1-5
Is an array that is in sorted order a min-heap?

6.1-6
Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12\rangle$ a max-heap?

6.1-7
Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2\rfloor + 1, \lfloor n/2\rfloor + 2, \ldots, n$.

6.2 Maintaining the heap property

In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY. Its inputs are an array A and an index i into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Figure 6.2 illustrates the action of MAX-HEAPIFY. At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ is determined, and its index is stored in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$, which causes node i and its

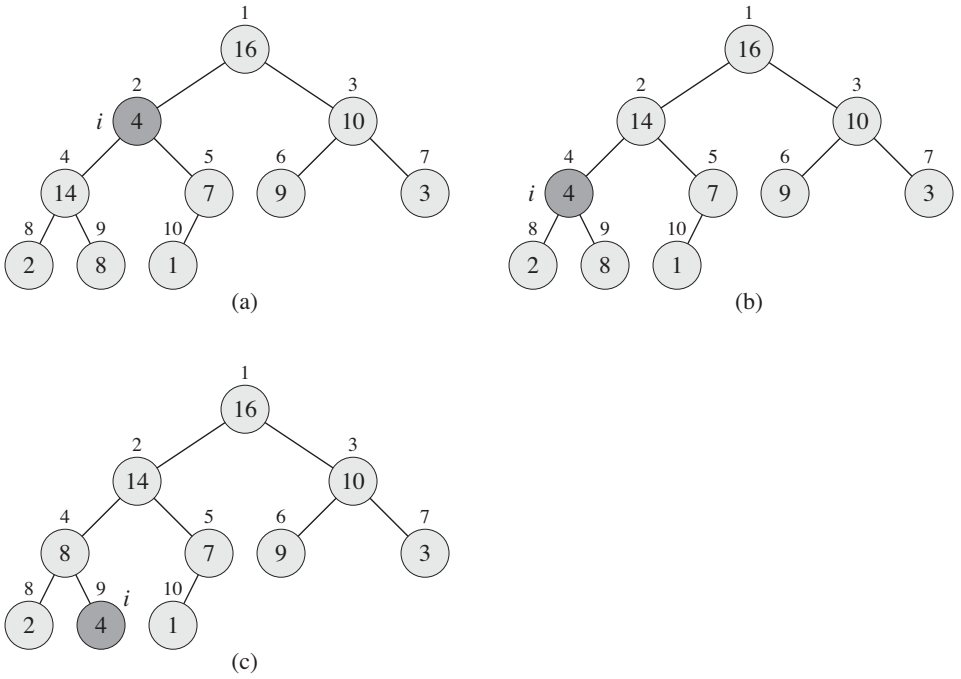


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

children to satisfy the max-heap property. The node indexed by *largest*, however, now has the original value $A[i]$, and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that subtree.

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1) .$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Exercises

6.2-1
Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.

6.2-2
Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

6.2-3
What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

6.2-4
What is the effect of calling MAX-HEAPIFY(A, i) for $i > A.\mathit{heap\text{-}size}/2$?

6.2-5
The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-6
Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

6.3 Building a heap

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array $A[1\,..\,n]$, where $n = A.\mathit{length}$, into a max-heap. By Exercise 6.1-7, the elements in the subarray $A[(\lfloor n/2\rfloor + 1)\,..\,n]$ are all leaves of the tree, and so each is

a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1$, $i + 2, \ldots, n$ is the root of a max-heap.

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: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-3).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, and so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by

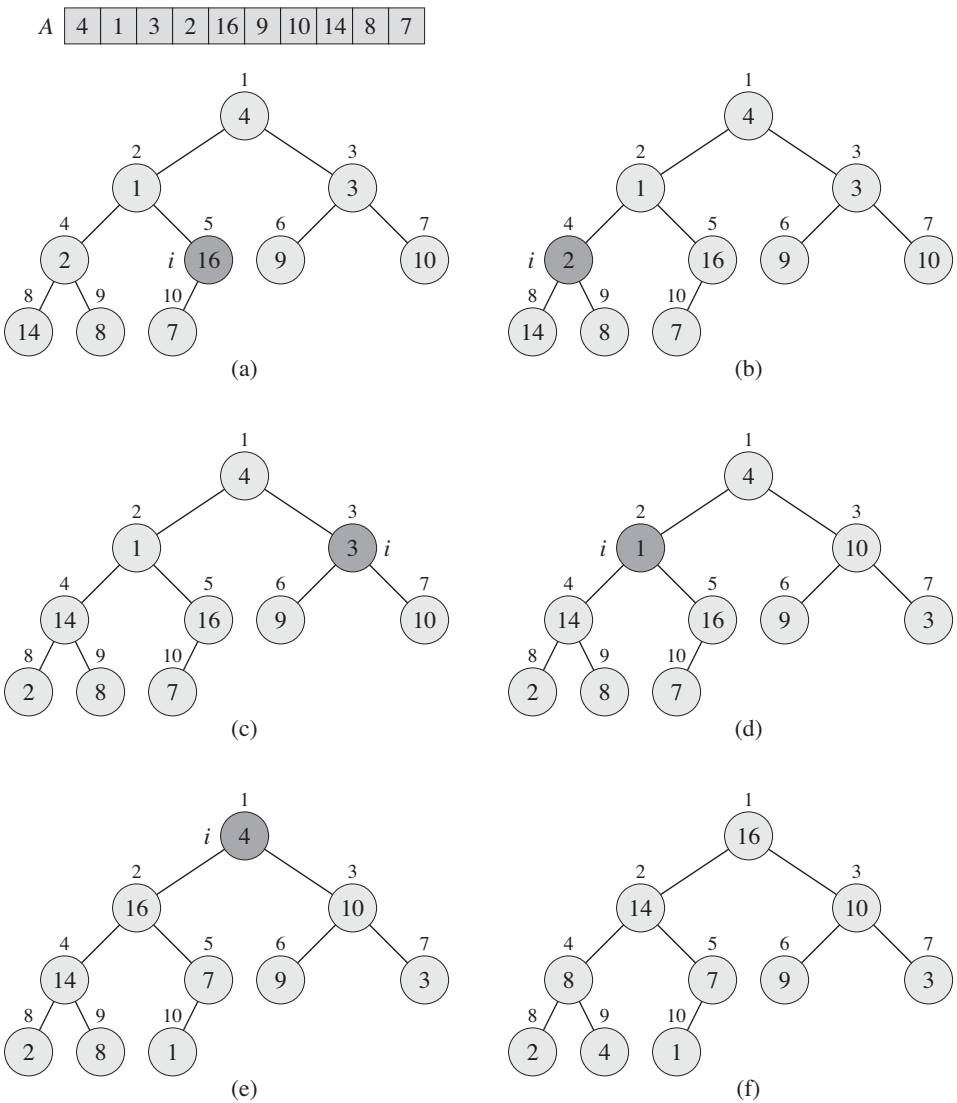


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

We evalaute the last summation by substituting $x = 1/2$ in the formula (A.8), yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, we can bound the running time of BUILD-MAX-HEAP as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

Hence, we can build a max-heap from an unordered array in linear time.

We can build a min-heap by the procedure BUILD-MIN-HEAP, which is the same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced by a call to MIN-HEAPIFY (see Exercise 6.2-2). BUILD-MIN-HEAP produces a min-heap from an unordered linear array in linear time.

Exercises

6.3-1
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2
Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

6.3-3
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

6.4 The heapsort algorithm

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1..n]$, where $n = A.length$. Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position

by exchanging it with $A[n]$. If we now discard node n from the heap—and we can do so by simply decrementing $A.heap\text{-}size$—we observe that the children of the root remain max-heaps, but the new root element might violate the max-heap property. All we need to do to restore the max-heap property, however, is call MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1\,..\,n-1]$. The heapsort algorithm then repeats this process for the max-heap of size $n-1$ down to a heap of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the **for** loop of lines 2–5 and after each iteration.

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n-1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

Exercises

6.4-1
Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2
Argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1\,..\,i]$ is a max-heap containing the i smallest elements of $A[1\,..\,n]$, and the subarray $A[i+1\,..\,n]$ contains the $n-i$ largest elements of $A[1\,..\,n]$, sorted.

6.4-3
What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

6.4-4
Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

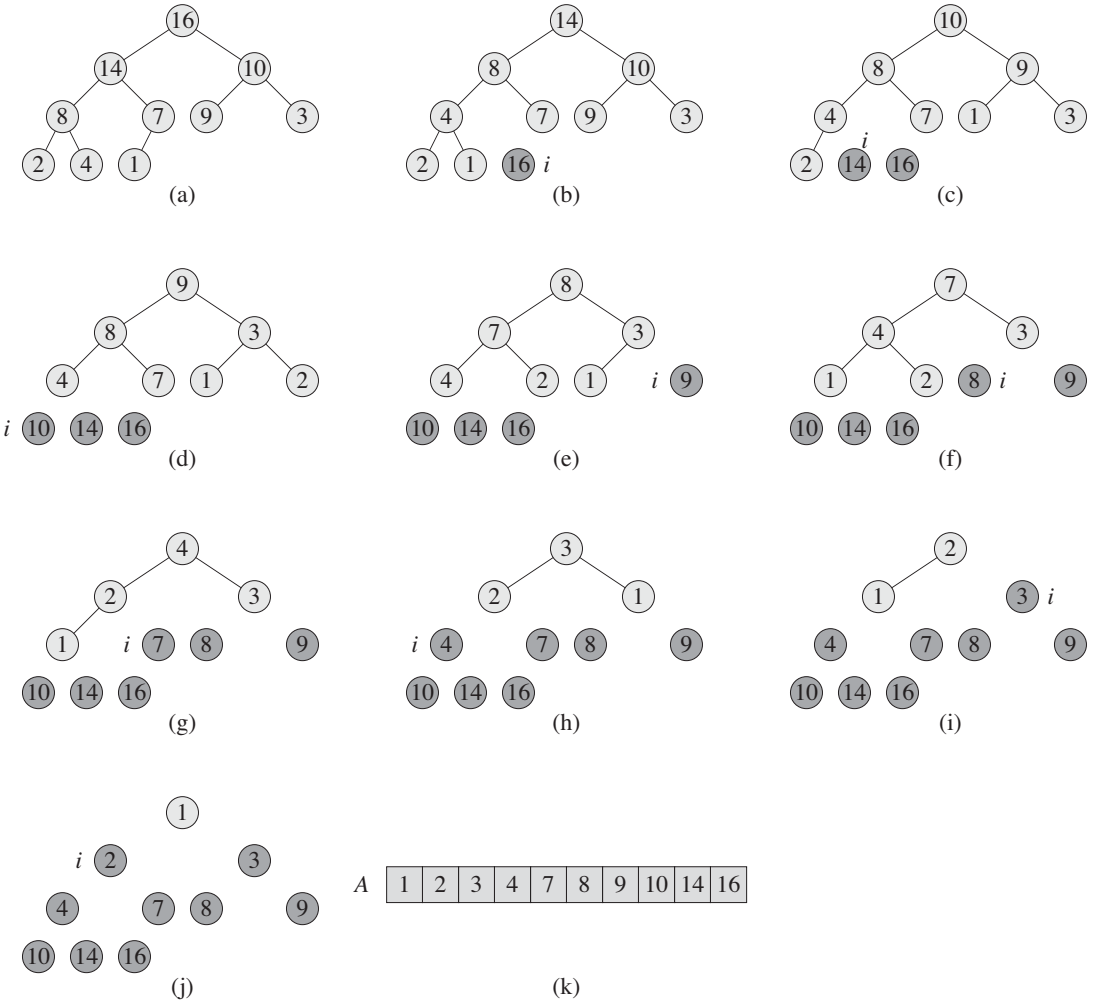


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

6.4-5 ⋆

Show that when all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

6.5 Priority queues

Heapsort is an excellent algorithm, but a good implementation of quicksort, presented in Chapter 7, usually beats it in practice. Nevertheless, the heap data structure itself has many uses. In this section, we present one of the most popular applications of a heap: as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We will focus here on how to implement max-priority queues, which are in turn based on max-heaps; Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Among their other applications, we can use max-priority queues to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX. The scheduler can add a new job to the queue at any time by calling INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, the simulator inserts them into the min-priority queue by calling INSERT.

We shall see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 23 and 24.

Not surprisingly, we can use a heap to implement a priority queue. In a given application, such as job scheduling or event-driven simulation, elements of a priority queue correspond to objects in the application. We often need to determine which application object corresponds to a given priority-queue element, and vice versa. When we use a heap to implement a priority queue, therefore, we often need to store a ***handle*** to the corresponding application object in each heap element. The exact makeup of the handle (such as a pointer or an integer) depends on the application. Similarly, we need to store a handle to the corresponding heap element in each application object. Here, the handle would typically be an array index. Because heap elements change locations within the array during heap operations, an actual implementation, upon relocating a heap element, would also have to update the array index in the corresponding application object. Because the details of accessing application objects depend heavily on the application and its implementation, we shall not pursue them here, other than noting that in practice, these handles do need to be correctly maintained.

Now we discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM implements the MAXIMUM operation in $\Theta(1)$ time.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

The running time of HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY.

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation. An index i into the array identifies the priority-queue element whose key we wish to increase. The procedure first updates the key of element $A[i]$ to its new value. Because increasing the key of $A[i]$ might violate the max-heap property,

the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT from Section 2.1, traverses a simple path from this node toward the root to find a proper place for the newly increased key. As HEAP-INCREASE-KEY traverses this path, it repeatedly compares an element to its parent, exchanging their keys and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds. (See Exercise 6.5-5 for a precise loop invariant.)

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

Figure 6.5 shows an example of a HEAP-INCREASE-KEY operation. The running time of HEAP-INCREASE-KEY on an n-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$.

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new element to be inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time.

Exercises

6.5-1

Illustrate the operation of HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

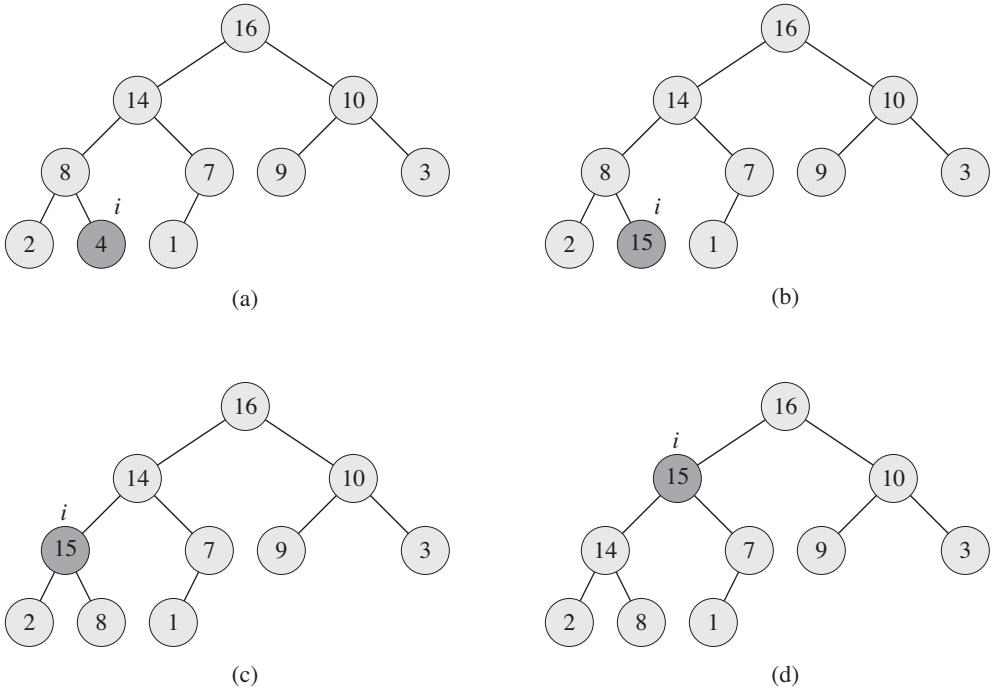


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

6.5-2

Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-3

Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

6.5-4

Why do we bother setting the key of the inserted node to $-\infty$ in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value?

6.5-5
Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:

> At the start of each iteration of the **while** loop of lines 4–6, the subarray $A[1 \,..\, A.heap\text{-}size]$ satisfies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.

You may assume that the subarray $A[1 \,..\, A.heap\text{-}size]$ satisfies the max-heap property at the time HEAP-INCREASE-KEY is called.

6.5-6
Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires three assignments. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments down to just one assignment.

6.5-7
Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.)

6.5-8
The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

6.5-9
Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* Use a min-heap for k-way merging.)

Problems

6-1 Building a heap using insertion
We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation on the BUILD-MAX-HEAP procedure:

```
BUILD-MAX-HEAP′(A)
1  A.heap-size = 1
2  for i = 2 to A.length
3      MAX-HEAP-INSERT(A, A[i])
```

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

6-2 Analysis of d-ary heaps

A ***d-ary heap*** is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INCREASE-KEY(A, i, k), which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

6-3 Young tableaus

An $m \times n$ ***Young tableau*** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \le mn$ finite numbers.

a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1, 1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m, n] < \infty$.

c. Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (*Hint:* Think about MAX-HEAPIFY.) Define $T(p)$, where $p = m + n$, to be the maximum running time of EXTRACT-MIN on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m + n)$ time bound.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

f. Give an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Chapter notes

The heapsort algorithm was invented by Williams [357], who also described how to implement a priority queue with a heap. The BUILD-MAX-HEAP procedure was suggested by Floyd [106].

We use min-heaps to implement min-priority queues in Chapters 16, 23, and 24. We also give an implementation with improved time bounds for certain operations in Chapter 19 and, assuming that the keys are drawn from a bounded set of nonnegative integers, Chapter 20.

If the data are b-bit integers, and the computer memory consists of addressable b-bit words, Fredman and Willard [115] showed how to implement MINIMUM in $O(1)$ time and INSERT and EXTRACT-MIN in $O(\sqrt{\lg n})$ time. Thorup [337] has improved the $O(\sqrt{\lg n})$ bound to $O(\lg \lg n)$ time. This bound uses an amount of space unbounded in n, but it can be implemented in linear space by using randomized hashing.

An important special case of priority queues occurs when the sequence of EXTRACT-MIN operations is ***monotone***, that is, the values returned by successive EXTRACT-MIN operations are monotonically increasing over time. This case arises in several important applications, such as Dijkstra's single-source shortest-paths algorithm, which we discuss in Chapter 24, and in discrete-event simulation. For Dijkstra's algorithm it is particularly important that the DECREASE-KEY operation be implemented efficiently. For the monotone case, if the data are integers in the range $1, 2, \ldots, C$, Ahuja, Mehlhorn, Orlin, and Tarjan [8] describe

how to implement EXTRACT-MIN and INSERT in $O(\lg C)$ amortized time (see Chapter 17 for more on amortized analysis) and DECREASE-KEY in $O(1)$ time, using a data structure called a radix heap. The $O(\lg C)$ bound can be improved to $O(\sqrt{\lg C})$ using Fibonacci heaps (see Chapter 19) in conjunction with radix heaps. Cherkassky, Goldberg, and Silverstein [65] further improved the bound to $O(\lg^{1/3+\epsilon} C)$ expected time by combining the multilevel bucketing structure of Denardo and Fox [85] with the heap of Thorup mentioned earlier. Raman [291] further improved these results to obtain a bound of $O(\min(\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n))$, for any fixed $\epsilon > 0$.

7 Quicksort

The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $\Theta(n \lg n)$, and the constant factors hidden in the $\Theta(n \lg n)$ notation are quite small. It also has the advantage of sorting in place (see page 17), and it works well even in virtual-memory environments.

Section 7.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we start with an intuitive discussion of its performance in Section 7.2 and postpone its precise analysis to the end of the chapter. Section 7.3 presents a version of quicksort that uses random sampling. This algorithm has a good expected running time, and no particular input elicits its worst-case behavior. Section 7.4 analyzes the randomized algorithm, showing that it runs in $\Theta(n^2)$ time in the worst case and, assuming distinct elements, in expected $O(n \lg n)$ time.

7.1 Description of quicksort

Quicksort, like merge sort, applies the divide-and-conquer paradigm introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \,..\, r]$:

Divide: Partition (rearrange) the array $A[p \,..\, r]$ into two (possibly empty) subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ such that each element of $A[p \,..\, q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \,..\, r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p\,..\,r]$ is now sorted.

The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p\,..\,r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION always selects an element $x = A[r]$ as a ***pivot*** element around which to partition the subarray $A[p\,..\,r]$. As the procedure runs, it partitions the array into four (possibly empty) regions. At the start of each iteration of the **for** loop in lines 3–6, the regions satisfy certain properties, shown in Figure 7.2. We state these properties as a loop invariant:

> At the beginning of each iteration of the loop of lines 3–6, for any array index k,
>
> 1. If $p \le k \le i$, then $A[k] \le x$.
> 2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
> 3. If $k = r$, then $A[k] = x$.

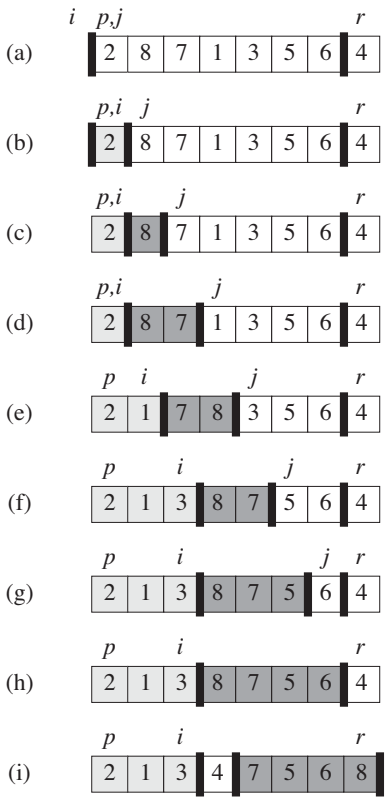


Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

The indices between j and $r - 1$ are not covered by any of the three cases, and the values in these entries have no particular relationship to the pivot x.

We need to show that this loop invariant is true prior to the first 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.

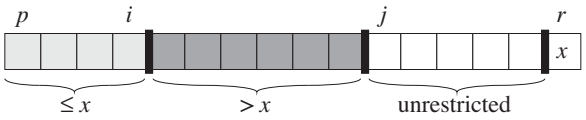


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p\,..\,r]$. The values in $A[p\,..\,i]$ are all less than or equal to x, the values in $A[i+1\,..\,j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j\,..\,r-1]$ can take on any values.

Initialization: Prior to the first iteration of the loop, $i = p-1$ and $j = p$. Because no values lie between p and i and no values lie between $i+1$ and $j-1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$; the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$; the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

The final two lines of PARTITION finish up by swapping the pivot element with the leftmost element greater than x, thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot's new index. The output of PARTITION now satisfies the specifications given for the divide step. In fact, it satisfies a slightly stronger condition: after line 2 of QUICKSORT, $A[q]$ is strictly less than every element of $A[q+1\,..\,r]$.

The running time of PARTITION on the subarray $A[p\,..\,r]$ is $\Theta(n)$, where $n = r - p + 1$ (see Exercise 7.1-3).

Exercises

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11\rangle$.

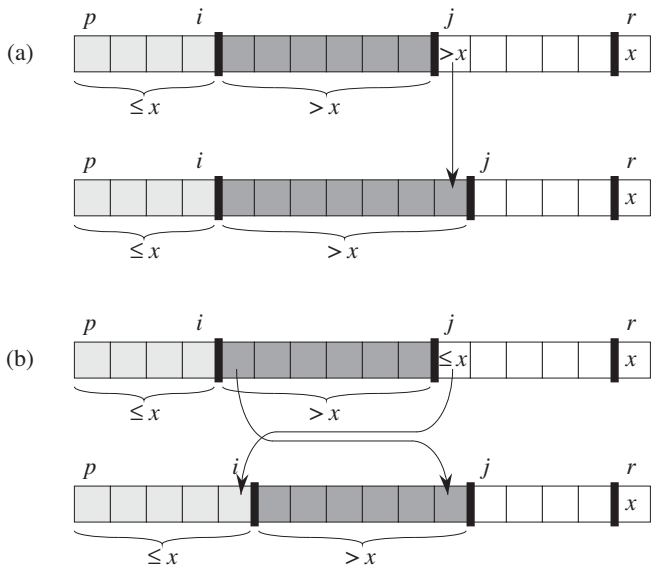


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

7.1-2

What value of q does PARTITION return when all elements in the array $A[p..r]$ have the same value? Modify PARTITION so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p..r]$ have the same value.

7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

7.1-4

How would you modify QUICKSORT to sort into nonincreasing order?

7.2 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge

sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. In this section, we shall informally investigate how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements. (We prove this claim in Section 7.4.1.) Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \,. \end{aligned}$$

Intuitively, if we sum the costs incurred at each level of the recursion, we get an arithmetic series (equation (A.2)), which evaluates to $\Theta(n^2)$. Indeed, it is straightforward to use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. (See Exercise 7.2-1.)

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted—a common situation in which insertion sort runs in $O(n)$ time.

Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this case, quicksort runs much faster. The recurrence for the running time is then

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

where we tolerate the sloppiness from ignoring the floor and ceiling and from subtracting 1. By case 2 of the master theorem (Theorem 4.1), this recurrence has the solution $T(n) = \Theta(n \lg n)$. By equally balancing the two sides of the partition at every level of the recursion, we get an asymptotically faster algorithm.

Balanced partitioning

The average-case running time of quicksort is much closer to the best case than to the worst case, as the analyses in Section 7.4 will show. The key to understand-

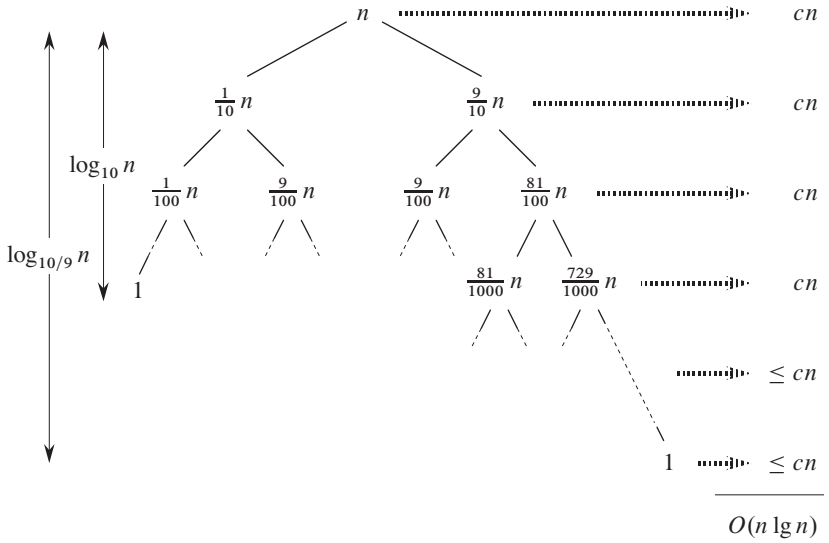


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

ing why is to understand how the balance of the partitioning is reflected in the recurrence that describes the running time.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) = T(9n/10) + T(n/10) + cn \ ,$$

on the running time of quicksort, where we have explicitly included the constant c hidden in the $\Theta(n)$ term. Figure 7.4 shows the recursion tree for this recurrence. Notice that every level of the tree has cost cn, until the recursion reaches a boundary condition at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most cn. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. The total cost of quicksort is therefore $O(n \lg n)$. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $O(n \lg n)$ time—asymptotically the same as if the split were right down the middle. Indeed, even a 99-to-1 split yields an $O(n \lg n)$ running time. In fact, any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \lg n)$ whenever the split has constant proportionality.

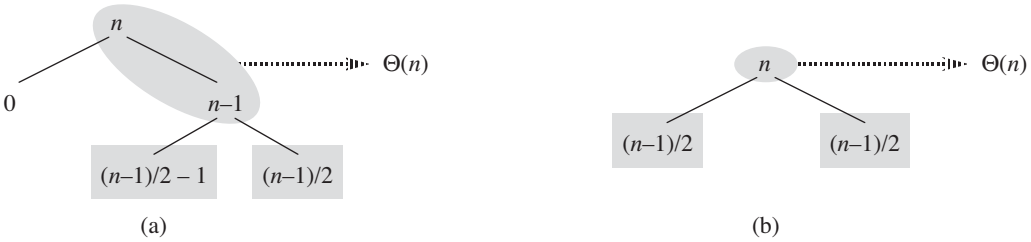


Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n-1$. The partitioning of the subarray of size $n-1$ costs $n-1$ and produces a "good" split: subarrays of size $(n-1)/2-1$ and $(n-1)/2$. **(b)** A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

Intuition for the average case

To develop a clear notion of the randomized behavior of quicksort, we must make an assumption about how frequently we expect to encounter the various inputs. The behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array. As in our probabilistic analysis of the hiring problem in Section 5.2, we will assume for now that all permutations of the input numbers are equally likely.

When we run quicksort on a random input array, the partitioning is highly unlikely to happen in the same way at every level, as our informal analysis has assumed. We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. For example, Exercise 7.2-6 asks you to show that about 80 percent of the time PARTITION produces a split that is more balanced than 9 to 1, and about 20 percent of the time it produces a split that is less balanced than 9 to 1.

In the average case, PARTITION produces a mix of "good" and "bad" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree. Suppose, for the sake of intuition, that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. Figure 7.5(a) shows the splits at two consecutive levels in the recursion tree. At the root of the tree, the cost is n for partitioning, and the subarrays produced have sizes $n-1$ and 0: the worst case. At the next level, the subarray of size $n-1$ undergoes best-case partitioning into subarrays of size $(n-1)/2-1$ and $(n-1)/2$. Let's assume that the boundary-condition cost is 1 for the subarray of size 0.

The combination of the bad split followed by the good split produces three subarrays of sizes 0, $(n-1)/2-1$, and $(n-1)/2$ at a combined partitioning cost of $\Theta(n)+\Theta(n-1)=\Theta(n)$. Certainly, this situation is no worse than that in Figure 7.5(b), namely a single level of partitioning that produces two subarrays of size $(n-1)/2$, at a cost of $\Theta(n)$. Yet this latter situation is balanced! Intuitively, the $\Theta(n-1)$ cost of the bad split can be absorbed into the $\Theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \lg n)$, but with a slightly larger constant hidden by the O-notation. We shall give a rigorous analysis of the expected running time of a randomized version of quicksort in Section 7.4.2.

Exercises

7.2-1
Use the substitution method to prove that the recurrence $T(n)=T(n-1)+\Theta(n)$ has the solution $T(n)=\Theta(n^2)$, as claimed at the beginning of Section 7.2.

7.2-2
What is the running time of QUICKSORT when all elements of array A have the same value?

7.2-3
Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

7.2-4
Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

7.2-5
Suppose that the splits at every level of quicksort are in the proportion $1-\alpha$ to α, where $0<\alpha\le 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n/\lg\alpha$ and the maximum depth is approximately $-\lg n/\lg(1-\alpha)$. (Don't worry about integer round-off.)

7.2-6 ⋆

Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that on a random input array, PARTITION produces a split more balanced than $1-\alpha$ to α.

7.3 A randomized version of quicksort

In exploring the average-case behavior of quicksort, we have made an assumption that all permutations of the input numbers are equally likely. In an engineering situation, however, we cannot always expect this assumption to hold. (See Exercise 7.2-4.) As we saw in Section 5.3, we can sometimes add randomization to an algorithm in order to obtain good expected performance over all inputs. Many people regard the resulting randomized version of quicksort as the sorting algorithm of choice for large enough inputs.

In Section 5.3, we randomized our algorithm by explicitly permuting the input. We could do so for quicksort also, but a different randomization technique, called ***random sampling***, yields a simpler analysis. Instead of always using $A[r]$ as the pivot, we will select a randomly chosen element from the subarray $A[p\,..\,r]$. We do so by first exchanging element $A[r]$ with an element chosen at random from $A[p\,..\,r]$. By randomly sampling the range $p, \ldots, r$, we ensure that the pivot element $x = A[r]$ is equally likely to be any of the $r - p + 1$ elements in the subarray. Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average.

The changes to PARTITION and QUICKSORT are small. In the new partition procedure, we simply implement the swap before actually partitioning:

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

We analyze this algorithm in the next section.

Exercises

7.3-1
Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

7.3-2
When RANDOMIZED-QUICKSORT runs, how many calls are made to the random-number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

7.4 Analysis of quicksort

Section 7.2 gave some intuition for the worst-case behavior of quicksort and for why we expect it to run quickly. In this section, we analyze the behavior of quicksort more rigorously. We begin with a worst-case analysis, which applies to either QUICKSORT or RANDOMIZED-QUICKSORT, and conclude with an analysis of the expected running time of RANDOMIZED-QUICKSORT.

7.4.1 Worst-case analysis

We saw in Section 7.2 that a worst-case split at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm. We now prove this assertion.

Using the substitution method (see Section 4.3), we can show that the running time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \tag{7.1}$$

where the parameter q ranges from 0 to $n-1$ because the procedure PARTITION produces two subproblems with total size $n-1$. We guess that $T(n) \le cn^2$ for some constant c. Substituting this guess into recurrence (7.1), we obtain

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

The expression $q^2 + (n-q-1)^2$ achieves a maximum over the parameter's range $0 \le q \le n-1$ at either endpoint. To verify this claim, note that the second derivative of the expression with respect to q is positive (see Exercise 7.4-3). This

observation gives us the bound $\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuing with our bounding of $T(n)$, we obtain

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

since we can pick the constant c large enough so that the $c(2n-1)$ term dominates the $\Theta(n)$ term. Thus, $T(n) = O(n^2)$. We saw in Section 7.2 a specific case in which quicksort takes $\Omega(n^2)$ time: when partitioning is unbalanced. Alternatively, Exercise 7.4-1 asks you to show that recurrence (7.1) has a solution of $T(n) = \Omega(n^2)$. Thus, the (worst-case) running time of quicksort is $\Theta(n^2)$.

7.4.2 Expected running time

We have already seen the intuition behind why the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$: if, in each level of recursion, the split induced by RANDOMIZED-PARTITION puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth $\Theta(\lg n)$, and $O(n)$ work is performed at each level. Even if we add a few new levels with the most unbalanced split possible between these levels, the total time remains $O(n \lg n)$. We can analyze the expected running time of RANDOMIZED-QUICKSORT precisely by first understanding how the partitioning procedure operates and then using this understanding to derive an $O(n \lg n)$ bound on the expected running time. This upper bound on the expected running time, combined with the $\Theta(n \lg n)$ best-case bound we saw in Section 7.2, yields a $\Theta(n \lg n)$ expected running time. We assume throughout that the values of the elements being sorted are distinct.

Running time and comparisons

The QUICKSORT and RANDOMIZED-QUICKSORT procedures differ only in how they select pivot elements; they are the same in all other respects. We can therefore couch our analysis of RANDOMIZED-QUICKSORT by discussing the QUICKSORT and PARTITION procedures, but with the assumption that pivot elements are selected randomly from the subarray passed to RANDOMIZED-PARTITION.

The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure. Each time the PARTITION procedure is called, it selects a pivot element, and this element is never included in any future recursive calls to QUICKSORT and PARTITION. Thus, there can be at most n calls to PARTITION over the entire execution of the quicksort algorithm. One call to PARTITION takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the **for** loop in lines 3–6. Each iteration of this **for** loop performs a comparison in line 4, comparing the pivot element to another element of the array A. Therefore,

if we can count the total number of times that line 4 is executed, we can bound the total time spent in the **for** loop during the entire execution of QUICKSORT.

Lemma 7.1
Let X be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

Proof By the discussion above, the algorithm makes at most n calls to PARTITION, each of which does a constant amount of work and then executes the **for** loop some number of times. Each iteration of the **for** loop executes line 4. ■

Our goal, therefore, is to compute X, the total number of comparisons performed in all calls to PARTITION. We will not attempt to analyze how many comparisons are made in *each* call to PARTITION. Rather, we will derive an overall bound on the total number of comparisons. To do so, we must understand when the algorithm compares two elements of the array and when it does not. For ease of analysis, we rename the elements of the array A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element. We also define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ to be the set of elements between z_i and z_j, inclusive.

When does the algorithm compare z_i and z_j? To answer this question, we first observe that each pair of elements is compared at most once. Why? Elements are compared only to the pivot element and, after a particular call of PARTITION finishes, the pivot element used in that call is never again compared to any other elements.

Our analysis uses indicator random variables (see Section 5.2). We define

$$X_{ij} = \mathrm{I}\{z_i \text{ is compared to } z_j\} ,$$

where we are considering whether the comparison takes place at any time during the execution of the algorithm, not just during one iteration or one call of PARTITION. Since each pair is compared at most once, we can easily characterize the total number of comparisons performed by the algorithm:

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

Taking expectations of both sides, and then using linearity of expectation and Lemma 5.1, we obtain

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \ . \tag{7.2}$$

It remains to compute $\Pr\{z_i \text{ is compared to } z_j\}$. Our analysis assumes that the RANDOMIZED-PARTITION procedure chooses each pivot randomly and independently.

Let us think about when two items are *not* compared. Consider an input to quicksort of the numbers 1 through 10 (in any order), and suppose that the first pivot element is 7. Then the first call to PARTITION separates the numbers into two sets: $\{1, 2, 3, 4, 5, 6\}$ and $\{8, 9, 10\}$. In doing so, the pivot element 7 is compared to all other elements, but no number from the first set (e.g., 2) is or ever will be compared to any number from the second set (e.g., 9).

In general, because we assume that element values are distinct, once a pivot x is chosen with $z_i < x < z_j$, we know that z_i and z_j cannot be compared at any subsequent time. If, on the other hand, z_i is chosen as a pivot before any other item in Z_{ij}, then z_i will be compared to each item in Z_{ij}, except for itself. Similarly, if z_j is chosen as a pivot before any other item in Z_{ij}, then z_j will be compared to each item in Z_{ij}, except for itself. In our example, the values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as a pivot. In contrast, 2 and 9 will never be compared because the first pivot element chosen from $Z_{2,9}$ is 7. Thus, z_i and z_j are compared if and only if the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j.

We now compute the probability that this event occurs. Prior to the point at which an element from Z_{ij} has been chosen as a pivot, the whole set Z_{ij} is together in the same partition. Therefore, any element of Z_{ij} is equally likely to be the first one chosen as a pivot. Because the set Z_{ij} has $j-i+1$ elements, and because pivots are chosen randomly and independently, the probability that any given element is the first one chosen as a pivot is $1/(j-i+1)$. Thus, we have

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} \ .
\end{aligned} \tag{7.3}$$

The second line follows because the two events are mutually exclusive. Combining equations (7.2) and (7.3), we get that

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

We can evaluate this sum using a change of variables ($k = j - i$) and the bound on the harmonic series in equation (A.7):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) . \qquad (7.4)
\end{aligned}$$

Thus we conclude that, using RANDOMIZED-PARTITION, the expected running time of quicksort is $O(n \lg n)$ when element values are distinct.

Exercises

7.4-1

Show that in the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) ,$$

$T(n) = \Omega(n^2)$.

7.4-2

Show that quicksort's best-case running time is $\Omega(n \lg n)$.

7.4-3

Show that the expression $q^2 + (n-q-1)^2$ achieves a maximum over $q = 0, 1, \ldots, n-1$ when $q = 0$ or $q = n-1$.

7.4-4

Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

7.4-5
We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should we pick k, both in theory and in practice?

7.4-6 ⋆
Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median (the middle value of the three elements). Approximate the probability of getting at worst an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

Problems

7-1 Hoare partition correctness
The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partition algorithm, which is due to C. A. R. Hoare:

```
HOARE-PARTITION(A, p, r)
 1  x = A[p]
 2  i = p − 1
 3  j = r + 1
 4  while TRUE
 5      repeat
 6          j = j − 1
 7      until A[j] ≤ x
 8      repeat
 9          i = i + 1
10      until A[i] ≥ x
11      if i < j
12          exchange A[i] with A[j]
13      else return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and auxiliary values after each iteration of the **while** loop in lines 4–13.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p..r]$ contains at least two elements, prove the following:

b. The indices i and j are such that we never access an element of A outside the subarray $A[p..r]$.

c. When HOARE-PARTITION terminates, it returns a value j such that $p \le j < r$.

d. Every element of $A[p..j]$ is less than or equal to every element of $A[j+1..r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in Section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p..j]$ and $A[j+1..r]$. Since $p \le j < r$, this split is always nontrivial.

e. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

7-2 Quicksort with equal element values

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

a. Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

b. The PARTITION procedure returns an index q such that each element of $A[p..q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1..r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p..r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that

- all elements of $A[q..t]$ are equal,
- each element of $A[p..q-1]$ is less than $A[q]$, and
- each element of $A[t+1..r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.

c. Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-QUICKSORT$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(p, r)$ that calls

RANDOMIZED-PARTITION′ and recurses only on partitions of elements not known to be equal to each other.

d. Using QUICKSORT′, how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?

7-3 Alternative quicksort analysis

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to RANDOMIZED-QUICKSORT, rather than on the number of comparisons performed.

a. Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables $X_i = \mathrm{I}\{i\text{th smallest element is chosen as the pivot}\}$. What is $\mathrm{E}[X_i]$?

b. Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] . \tag{7.5}$$

c. Show that we can rewrite equation (7.5) as

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}[T(q)] + \Theta(n) . \tag{7.6}$$

d. Show that

$$\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \tag{7.7}$$

(*Hint:* Split the summation into two parts, one for $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$ and one for $k = \lceil n/2 \rceil, \ldots, n-1$.)

e. Using the bound from equation (7.7), show that the recurrence in equation (7.6) has the solution $\mathrm{E}[T(n)] = \Theta(n \lg n)$. (*Hint:* Show, by substitution, that $\mathrm{E}[T(n)] \le an \lg n$ for sufficiently large n and for some positive constant a.)

7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

a. Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ correctly sorts the array A.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

7-5 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the ***median-of-3*** method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, let us assume that the elements in the input array $A[1..n]$ are distinct and that $n \geq 3$. We denote the

sorted output array by $A'[1..n]$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = A'[i]\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n-1$. (Note that $p_1 = p_n = 0$.)

b. By what amount have we increased the likelihood of choosing the pivot as $x = A'[\lfloor (n+1)/2 \rfloor]$, the median of $A[1..n]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

c. If we define a "good" split to mean choosing the pivot as $x = A'[i]$, where $n/3 \le i \le 2n/3$, by what amount have we increased the likelihood of getting a good split compared with the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

7-6 Fuzzy sorting of intervals

Consider a sorting problem in which we do not know the numbers exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given n closed intervals of the form $[a_i, b_i]$, where $a_i \le b_i$. We wish to ***fuzzy-sort*** these intervals, i.e., to produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that for $j = 1, 2, \ldots, n$, there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \le c_2 \le \cdots \le c_n$.

a. Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

b. Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

Chapter notes

The quicksort procedure was invented by Hoare [170]; Hoare's version appears in Problem 7-1. The PARTITION procedure given in Section 7.1 is due to N. Lomuto. The analysis in Section 7.4 is due to Avrim Blum. Sedgewick [305] and Bentley [43] provide a good reference on the details of implementation and how they matter.

McIlroy [248] showed how to engineer a "killer adversary" that produces an array on which virtually any implementation of quicksort takes $\Theta(n^2)$ time. If the implementation is randomized, the adversary produces the array after seeing the random choices of the quicksort algorithm.

8 Sorting in Linear Time

We have now introduced several algorithms that can sort n numbers in $O(n \lg n)$ time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements*. We call such sorting algorithms ***comparison sorts***. All the sorting algorithms introduced thus far are comparison sorts.

In Section 8.1, we shall prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time. Of course, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

8.1 Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

In this section, we assume without loss of generality that all the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that

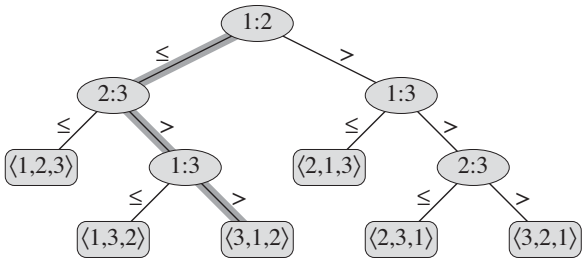


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \leq a_j$.

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A ***decision tree*** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

In a decision tree, we annotate each internal node by $i{:}j$ for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence. We also annotate each leaf by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (See Section C.1 for background on permutations.) The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \leq a_j$. The left subtree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons knowing that $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual

execution of the comparison sort. (We shall refer to such leaves as "reachable.") Thus, we shall consider only decision trees in which each permutation appears as a reachable leaf.

A lower bound for the worst case

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \leq l \leq 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\geq \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.19))} . \end{aligned}$$

■

Corollary 8.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

Exercises

8.1-1
What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

8.1-2
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

8.1-3
Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

8.1-4
Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (*Hint:* It is not rigorous to simply combine the lower bounds for the individual subsequences.)

8.2 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than x, then x belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $A.length = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[0..k]$ provides temporary working storage.

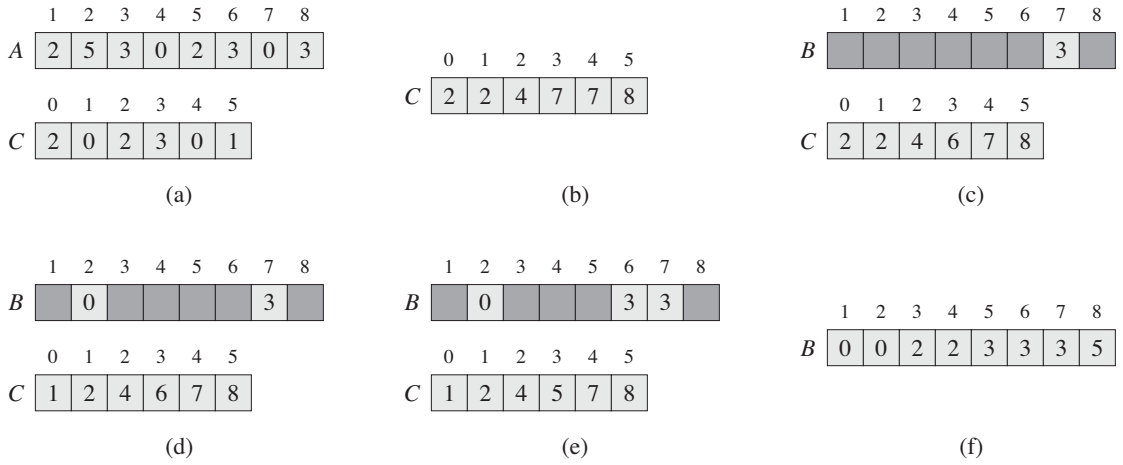


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array C to all zeros, the **for** loop of lines 4–5 inspects each input element. If the value of an input element is i, we increment $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C.

Finally, the **for** loop of lines 10–12 places each element $A[j]$ into its correct sorted position in the output array B. If all n elements are distinct, then when we first enter line 10, for each $A[j]$, the value $C[A[j]]$ is the correct final position of $A[j]$ in the output array, since there are $C[A[j]]$ elements less than or equal to $A[j]$. Because the elements might not be distinct, we decrement $C[A[j]]$ each time we place a value $A[j]$ into the B array. Decrementing $C[A[j]]$ causes the next input element with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array.

How much time does counting sort require? The **for** loop of lines 2–3 takes time $\Theta(k)$, the **for** loop of lines 4–5 takes time $\Theta(n)$, the **for** loop of lines 7–8 takes time $\Theta(k)$, and the **for** loop of lines 10–12 takes time $\Theta(n)$. Thus, the overall time is $\Theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Counting sort beats the lower bound of $\Omega(n \lg n)$ proved in Section 8.1 because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison sort model.

An important property of counting sort is that it is ***stable***: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Exercises

8.2-1

Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2

Prove that COUNTING-SORT is stable.

8.2-3

Suppose that we were to rewrite the **for** loop header in line 10 of the COUNTING-SORT as

```
10  for j = 1 to A.length
```

Show that the algorithm still works properly. Is the modified algorithm stable?

8.2-4
Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a\,..\,b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

8.3 Radix sort

Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A d-digit number would then occupy a field of d columns. Since the card sorter can look at only one column at a time, the problem of sorting n cards on a d-digit number requires a sorting algorithm.

Intuitively, you might sort numbers on their *most significant* digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of. (See Exercise 8.3-5.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the *least significant* digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner. The process continues until the cards have been sorted on all d digits. Remarkably, at that point the cards are fully sorted on the d-digit number. Thus, only d passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a "deck" of seven 3-digit numbers.

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator has to be wary about not changing the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

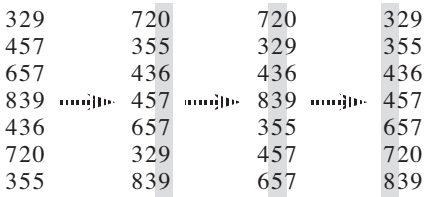


Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

RADIX-SORT(A, d)
1 **for** $i = 1$ **to** d
2 use a stable sort to sort array A on digit i

Lemma 8.3
Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

Proof The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to $k-1$ (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\Theta(n+k)$. There are d passes, and so the total time for radix sort is $\Theta(d(n + k))$. ■

When d is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

Lemma 8.4
Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n + 2^r))$ time if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to k.

Proof For a value $r \leq b$, we view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes time $\Theta(n + k) = \Theta(n + 2^r)$ and there are d passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

For given values of n and b, we wish to choose the value of r, with $r \leq b$, that minimizes the expression $(b/r)(n + 2^r)$. If $b < \lfloor \lg n \rfloor$, then for any value of $r \leq b$, we have that $(n + 2^r) = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $(b/b)(n + 2^b) = \Theta(n)$, which is asymptotically optimal. If $b \geq \lfloor \lg n \rfloor$, then choosing $r = \lfloor \lg n \rfloor$ gives the best time to within a constant factor, which we can see as follows. Choosing $r = \lfloor \lg n \rfloor$ yields a running time of $\Theta(bn/\lg n)$. As we increase r above $\lfloor \lg n \rfloor$, the 2^r term in the numerator increases faster than the r term in the denominator, and so increasing r above $\lfloor \lg n \rfloor$ yields a running time of $\Omega(bn/\lg n)$. If instead we were to decrease r below $\lfloor \lg n \rfloor$, then the b/r term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort? If $b = O(\lg n)$, as is often the case, and we choose $r \approx \lg n$, then radix sort's running time is $\Theta(n)$, which appears to be better than quicksort's expected running time of $\Theta(n \lg n)$. The constant factors hidden in the Θ-notation differ, however. Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer. Which sorting algorithm we prefer depends on the characteristics of the implementations, of the underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort), and of the input data. Moreover, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the $\Theta(n \lg n)$-time comparison sorts do. Thus, when primary memory storage is at a premium, we might prefer an in-place algorithm such as quicksort.

Exercises

8.3-1
Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3-2
Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

8.3-3
Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

8.3-4
Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

8.3-5 ⋆
In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

8.4 Bucket sort

Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$. (See Section C.2 for a definition of uniform distribution.)

Bucket sort divides the interval $[0, 1)$ into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Our code for bucket sort assumes that the input is an n-element array A and that each element $A[i]$ in the array satisfies $0 \leq A[i] < 1$. The code requires an auxiliary array $B[0..n-1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.)

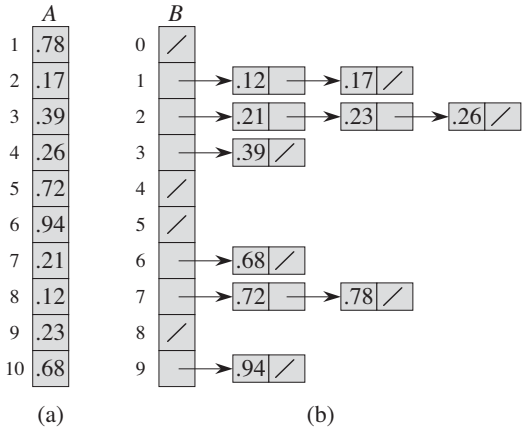


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

BUCKET-SORT(A)

```
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Since $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index. If $A[i]$ and $A[j]$ go into the same bucket, then the **for** loop of lines 7–8 puts them into the proper order. If $A[i]$ and $A[j]$ go into different buckets, then line 9 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that all lines except line 8 take $O(n)$ time in the worst case. We need to analyze the total time taken by the n calls to insertion sort in line 8.

To analyze the cost of the calls to insertion sort, let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time (see Section 2.2), the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \ .$$

We now analyze the average-case running time of bucket sort, by computing the expected value of the running time, where we take the expectation over the input distribution. Taking expectations of both sides and using linearity of expectation, we have

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(by linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(by equation (C.22))} \ . \qquad (8.1)
\end{aligned}$$

We claim that

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \qquad (8.2)$$

for $i = 0, 1, \ldots, n-1$. It is no surprise that each bucket i has the same value of $\mathrm{E}[n_i^2]$, since each value in the input array A is equally likely to fall in any bucket. To prove equation (8.2), we define indicator random variables

$$X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$$

for $i = 0, 1, \ldots, n-1$ and $j = 1, 2, \ldots, n$. Thus,

$$n_i = \sum_{j=1}^{n} X_{ij} \ .$$

To compute $\mathrm{E}[n_i^2]$, we expand the square and regroup terms:

$$
\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} \mathrm{E}\left[X_{ij}X_{ik}\right] , \qquad (8.3)
\end{aligned}
$$

where the last line follows by linearity of expectation. We evaluate the two summations separately. Indicator random variable X_{ij} is 1 with probability $1/n$ and 0 otherwise, and therefore

$$
\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) \\
&= \frac{1}{n} .
\end{aligned}
$$

When $k \ne j$, the variables X_{ij} and X_{ik} are independent, and hence

$$
\begin{aligned}
\mathrm{E}\left[X_{ij}X_{ik}\right] &= \mathrm{E}\left[X_{ij}\right]\mathrm{E}\left[X_{ik}\right] \\
&= \frac{1}{n}\cdot\frac{1}{n} \\
&= \frac{1}{n^2} .
\end{aligned}
$$

Substituting these two expected values in equation (8.3), we obtain

$$
\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \sum_{j=1}^{n}\frac{1}{n} + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} \frac{1}{n^2} \\
&= n\cdot\frac{1}{n} + n(n-1)\cdot\frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n} ,
\end{aligned}
$$

which proves equation (8.2).

Using this expected value in equation (8.1), we conclude that the average-case running time for bucket sort is $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$.

Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, equation (8.1) tells us that bucket sort will run in linear time.

Exercises

8.4-1

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

8.4-2

Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

8.4-3

Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $\mathrm{E}[X^2]$? What is $\mathrm{E}^2[X]$?

8.4-4 ⋆

We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

8.4-5 ⋆

A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that we draw a list of n random variables $X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time. Give an algorithm that sorts these numbers in linear average-case time.

Problems

8-1 Probabilistic lower bounds on comparison sorting

In this problem, we prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. We begin by examining a deterministic comparison sort A with decision tree T_A. We assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a decision tree T; that is, $D(T)$ is the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.

c. Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$. (*Hint:* Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ the number of leaves in RT.)

d. Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $i \lg i + (k-i) \lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.

Now, consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form $\text{RANDOM}(1, r)$ made by algorithm B; the node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B.

8-2 Sorting in place in linear time

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (*Hint:* How would you do it for $k = 3$?)

8-3 Sorting variable-length items

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time.

(Note that the desired order here is the standard alphabetical order; for example, `a` < `ab` < `b`.)

8-4 Water jugs

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.

Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k-sorted*** if, for all $i = 1, 2, \ldots, n-k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if $A[i] \leq A[i+k]$ for all $i = 1, 2, \ldots, n-k$.

d. Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show that we can sort a k-sorted array of length n in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-9.)

f. Show that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time. (*Hint:* Use the solution to the previous part along with the lower bound on comparison sorts.)

8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, we will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

First we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

a. Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now we will show a slightly tighter $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

d. Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

8-7 The 0-1 sorting lemma and columnsort

A ***compare-exchange*** operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form

```
COMPARE-EXCHANGE(A, i, j)
1  if A[i] > A[j]
2      exchange A[i] with A[j]
```

After the compare-exchange operation, we know that $A[i] \leq A[j]$.

An ***oblivious compare-exchange algorithm*** operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, here is insertion sort expressed as an oblivious compare-exchange algorithm:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      for i = j − 1 downto 1
3          COMPARE-EXCHANGE(A, i, i + 1)
```

The ***0-1 sorting lemma*** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array $A[1\,..\,n]$. Let $A[p]$ be the smallest value in A that algorithm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1\,..\,n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p]\,, \\ 1 & \text{if } A[i] > A[p]\,. \end{cases}$$

a. Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$.

b. To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array B correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, ***columnsort***, works on a rectangular array of n elements. The array has r rows and s columns (so that $n = rs$), subject to three restrictions:

- r must be even,
- s must be a divisor of r, and
- $r \ge 2s^2$.

When columnsort completes, the array is sorted in ***column-major order***: reading down the columns, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to r rows and s columns. In other words, turn the leftmost column into the top r/s rows, in order; turn the next column into the next r/s rows, in order; and so on.
3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.

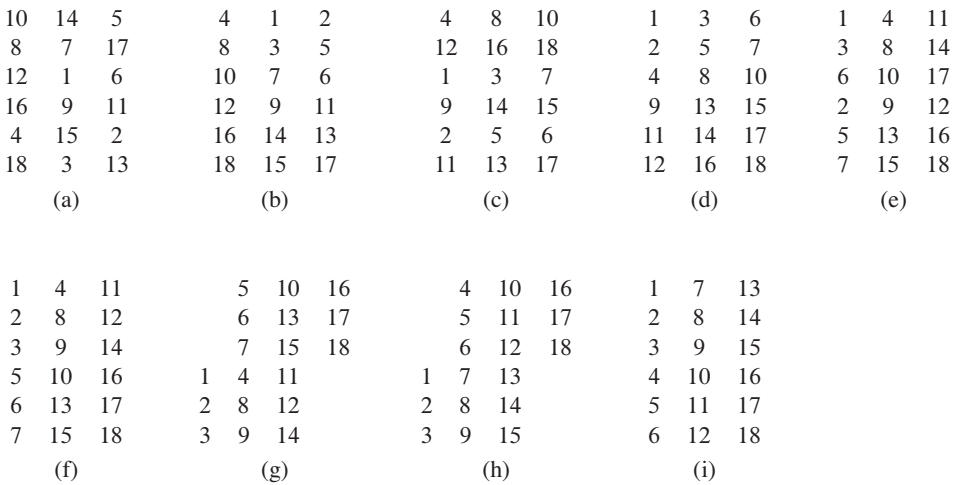


Figure 8.5 The steps of columnsort. **(a)** The input array with 6 rows and 3 columns. **(b)** After sorting each column in step 1. **(c)** After transposing and reshaping in step 2. **(d)** After sorting each column in step 3. **(e)** After performing step 4, which inverts the permutation from step 2. **(f)** After sorting each column in step 5. **(g)** After shifting by half a column in step 6. **(h)** After sorting each column in step 7. **(i)** After performing step 8, which inverts the permutation from step 6. The array is now sorted in column-major order.

5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

Figure 8.5 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. (Even though this example violates the requirement that $r \geq 2s^2$, it happens to work.)

c. Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A

couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with r rows and s columns.

d. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.

e. Prove that after step 4, the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle.

f. Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

g. Now suppose that s does not divide r. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?

h. Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [110]. Knuth's comprehensive treatise on sorting [211] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Ben-Or [39] studied lower bounds for sorting using generalizations of the decision-tree model.

Knuth credits H. H. Seward with inventing counting sort in 1954, as well as with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by E. J. Isaac and R. C. Singleton [188].

Munro and Raman [263] give a stable sorting algorithm that performs $O(n^{1+\epsilon})$ comparisons in the worst case, where $0 < \epsilon \leq 1$ is any fixed constant. Although

any of the $O(n \lg n)$-time algorithms make fewer comparisons, the algorithm by Munro and Raman moves data only $O(n)$ times and operates in place.

The case of sorting n b-bit integers in $o(n \lg n)$ time has been considered by many researchers. Several positive results have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm. All the results assume that the computer memory is divided into addressable b-bit words. Fredman and Willard [115] introduced the fusion tree data structure and used it to sort n integers in $O(n \lg n / \lg \lg n)$ time. This bound was later improved to $O(n \sqrt{\lg n})$ time by Andersson [16]. These algorithms require the use of multiplication and several precomputed constants. Andersson, Hagerup, Nilsson, and Raman [17] have shown how to sort n integers in $O(n \lg \lg n)$ time without using multiplication, but their method requires storage that can be unbounded in terms of n. Using multiplicative hashing, we can reduce the storage needed to $O(n)$, but then the $O(n \lg \lg n)$ worst-case bound on the running time becomes an expected-time bound. Generalizing the exponential search trees of Andersson [16], Thorup [335] gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and it uses linear space. Combining these techniques with some new ideas, Han [158] improved the bound for sorting to $O(n \lg \lg n \lg \lg \lg n)$ time. Although these algorithms are important theoretical breakthroughs, they are all fairly complicated and at the present time seem unlikely to compete with existing sorting algorithms in practice.

The columnsort algorithm in Problem 8-7 is by Leighton [227].

9 Medians and Order Statistics

The ith ***order statistic*** of a set of n elements is the ith smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the nth order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at $i = (n + 1)/2$. When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2+1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n + 1)/2 \rfloor$ (the ***lower median***) and $i = \lceil (n + 1)/2 \rceil$ (the ***upper median***). For simplicity in this text, however, we consistently use the phrase "the median" to refer to the lower median.

This chapter addresses the problem of selecting the ith order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. We formally specify the ***selection problem*** as follows:

Input: A set A of n (distinct) numbers and an integer i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

We can solve the selection problem in $O(n \lg n)$ time, since we can sort the numbers using heapsort or merge sort and then simply index the ith element in the output array. This chapter presents faster algorithms.

In Section 9.1, we examine the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which we investigate in the subsequent two sections. Section 9.2 analyzes a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming distinct elements. Section 9.3 contains an algorithm of more theoretical interest that achieves the $O(n)$ running time in the worst case.

9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n elements? We can easily obtain an upper bound of $n - 1$ comparisons: examine each element of the set in turn and keep track of the smallest element seen so far. In the following procedure, we assume that the set resides in array A, where $A.length = n$.

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

We can, of course, find the maximum with $n - 1$ comparisons as well.

Is this the best we can do? Yes, since we can obtain a lower bound of $n - 1$ comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. Observing that every element except the winner must lose at least one match, we conclude that $n - 1$ comparisons are necessary to determine the minimum. Hence, the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

Simultaneous minimum and maximum

In some applications, we must find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum value of each coordinate.

At this point, it should be obvious how to determine both the minimum and the maximum of n elements using $\Theta(n)$ comparisons, which is asymptotically optimal: simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons.

In fact, we can find both the minimum and the maximum using at most $3\lfloor n/2 \rfloor$ comparisons. We do so by maintaining both the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element,

we process elements in pairs. We compare pairs of elements from the input first *with each other*, and then we compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How we set up initial values for the current minimum and maximum depends on whether n is odd or even. If n is odd, we set both the minimum and maximum to the value of the first element, and then we process the rest of the elements in pairs. If n is even, we perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd n.

Let us analyze the total number of comparisons. If n is odd, then we perform $3\lfloor n/2 \rfloor$ comparisons. If n is even, we perform 1 initial comparison followed by $3(n-2)/2$ comparisons, for a total of $3n/2 - 2$. Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

Exercises

9.1-1
Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

9.1-2 ⋆
Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers. (*Hint:* Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

9.2 Selection in expected linear time

The general selection problem appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$. In this section, we present a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7. As in quicksort, we partition the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected running time of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct.

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 7.3. Thus, like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The following code for RANDOMIZED-SELECT returns the ith smallest element of the array $A[p..r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q - p + 1
5  if i == k            // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q - 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

The RANDOMIZED-SELECT procedure works as follows. Line 1 checks for the base case of the recursion, in which the subarray $A[p..r]$ consists of just one element. In this case, i must equal 1, and we simply return $A[p]$ in line 2 as the ith smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3 partitions the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which in turn is less than each element of $A[q+1..r]$. As in quicksort, we will refer to $A[q]$ as the ***pivot*** element. Line 4 computes the number k of elements in the subarray $A[p..q]$, that is, the number of elements in the low side of the partition, plus one for the pivot element. Line 5 then checks whether $A[q]$ is the ith smallest element. If it is, then line 6 returns $A[q]$. Otherwise, the algorithm determines in which of the two subarrays $A[p..q-1]$ and $A[q+1..r]$ the ith smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and line 8 recursively selects it from the subarray. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the ith smallest element of $A[p..r]$—namely, the elements of $A[p..q]$—the desired element is the $(i-k)$th smallest element of $A[q+1..r]$, which line 9 finds recursively. The code appears to allow recursive calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element, and partitioning takes $\Theta(n)$ time. We will see that

the algorithm has a linear expected running time, though, and because it is randomized, no particular input elicits the worst-case behavior.

To analyze the expected running time of RANDOMIZED-SELECT, we let the running time on an input array $A[p..r]$ of n elements be a random variable that we denote by $T(n)$, and we obtain an upper bound on $\mathrm{E}[T(n)]$ as follows. The procedure RANDOMIZED-PARTITION is equally likely to return any element as the pivot. Therefore, for each k such that $1 \le k \le n$, the subarray $A[p..q]$ has k elements (all less than or equal to the pivot) with probability $1/n$. For $k = 1, 2, \ldots, n$, we define indicator random variables X_k where

$$X_k = \mathrm{I}\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\} ,$$

and so, assuming that the elements are distinct, we have

$$\mathrm{E}[X_k] = 1/n . \tag{9.1}$$

When we call RANDOMIZED-SELECT and choose $A[q]$ as the pivot element, we do not know, a priori, if we will terminate immediately with the correct answer, recurse on the subarray $A[p..q-1]$, or recurse on the subarray $A[q+1..r]$. This decision depends on where the ith smallest element falls relative to $A[q]$. Assuming that $T(n)$ is monotonically increasing, we can upper-bound the time needed for the recursive call by the time needed for the recursive call on the largest possible input. In other words, to obtain an upper bound, we assume that the ith element is always on the side of the partition with the greater number of elements. For a given call of RANDOMIZED-SELECT, the indicator random variable X_k has the value 1 for exactly one value of k, and it is 0 for all other k. When $X_k = 1$, the two subarrays on which we might recurse have sizes $k-1$ and $n-k$. Hence, we have the recurrence

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) . \end{aligned}$$

Taking expected values, we have

$$\begin{aligned}
&\mathrm{E}[T(n)] \\
&\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(by linearity of expectation)} \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (C.24))} \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (9.1))} \ .
\end{aligned}$$

In order to apply equation (C.24), we rely on X_k and $T(\max(k-1, n-k))$ being independent random variables. Exercise 9.2-2 asks you to justify this assertion.

Let us consider the expression $\max(k-1, n-k)$. We have

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \ , \\ n-k & \text{if } k \le \lceil n/2 \rceil \ . \end{cases}$$

If n is even, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears exactly twice in the summation, and if n is odd, all these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once. Thus, we have

$$\mathrm{E}[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n) \ .$$

We show that $\mathrm{E}[T(n)] = O(n)$ by substitution. Assume that $\mathrm{E}[T(n)] \le cn$ for some constant c that satisfies the initial conditions of the recurrence. We assume that $T(n) = O(1)$ for n less than some constant; we shall pick this constant later. We also pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded from above by an for all $n > 0$. Using this inductive hypothesis, we have

$$\begin{aligned}
\mathrm{E}[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an
\end{aligned}$$

$$
\begin{aligned}
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor - 1)\,\lfloor n/2\rfloor}{2}\right) + an \\
&\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\
&= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\
&= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\
&\le \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) .
\end{aligned}
$$

In order to complete the proof, we need to show that for sufficiently large n, this last expression is at most cn or, equivalently, that $cn/4 - c/2 - an \ge 0$. If we add $c/2$ to both sides and factor out n, we get $n(c/4 - a) \ge c/2$. As long as we choose the constant c so that $c/4 - a > 0$, i.e., $c > 4a$, we can divide both sides by $c/4 - a$, giving

$$n \ge \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a} .$$

Thus, if we assume that $T(n) = O(1)$ for $n < 2c/(c-4a)$, then $\mathrm{E}[T(n)] = O(n)$. We conclude that we can find any order statistic, and in particular the median, in expected linear time, assuming that the elements are distinct.

Exercises

9.2-1

Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

9.2-2

Argue that the indicator random variable X_k and the value $T(\max(k-1, n-k))$ are independent.

9.2-3

Write an iterative version of RANDOMIZED-SELECT.

9.2-4

Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

9.3 Selection in worst-case linear time

We now examine a selection algorithm whose running time is $O(n)$ in the worst case. Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. Here, however, we *guarantee* a good split upon partitioning the array. SELECT uses the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), but modified to take the element to partition around as an input parameter.

The SELECT algorithm determines the ith smallest of an input array of $n > 1$ distinct elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the ith smallest.)

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are $n-k$ elements on the high side of the partition.
5. If $i = k$, then return x. Otherwise, use SELECT recursively to find the ith smallest element on the low side if $i < k$, or the $(i-k)$th smallest element on the high side if $i > k$.

To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element x. Figure 9.1 helps us to visualize this bookkeeping. At least half of the medians found in

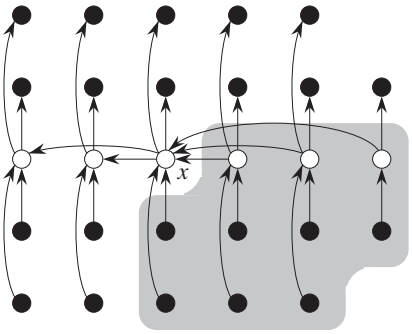


Figure 9.1 Analysis of the algorithm SELECT. The n elements are represented by small circles, and each group of 5 elements occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. (When finding the median of an even number of elements, we use the lower median.) Arrows go from larger elements to smaller, from which we can see that 3 out of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x. The elements known to be greater than x appear on a shaded background.

step 2 are greater than or equal to the median-of-medians x.[1] Thus, at least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x, except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \;\geq\; \frac{3n}{10} - 6\,.$$

Similarly, at least $3n/10 - 6$ elements are less than x. Thus, in the worst case, step 5 calls SELECT recursively on at most $7n/10 + 6$ elements.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$.) Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10 + 6)$, assuming that T is monotonically increasing. We make the assumption, which seems unmotivated at first, that any input of fewer than 140 elements requires $O(1)$ time; the origin of the magic constant 140 will be clear shortly. We can therefore obtain the recurrence

[1] Because of our assumption that the numbers are distinct, all medians except x are either greater than or less than x.

$$T(n) \le \begin{cases} O(1) & \text{if } n < 140 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \ge 140 . \end{cases}$$

We show that the running time is linear by substitution. More specifically, we will show that $T(n) \le cn$ for some suitably large constant c and all $n > 0$. We begin by assuming that $T(n) \le cn$ for some suitably large constant c and all $n < 140$; this assumption holds if c is large enough. We also pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by an for all $n > 0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) , \end{aligned}$$

which is at most cn if

$$-cn/10 + 7c + an \le 0 . \tag{9.2}$$

Inequality (9.2) is equivalent to the inequality $c \ge 10a(n/(n-70))$ when $n > 70$. Because we assume that $n \ge 140$, we have $n/(n-70) \le 2$, and so choosing $c \ge 20a$ will satisfy inequality (9.2). (Note that there is nothing special about the constant 140; we could replace it by any integer strictly greater than 70 and then choose c accordingly.) The worst-case running time of SELECT is therefore linear.

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input. They are not subject to the $\Omega(n \lg n)$ lower bound because they manage to solve the selection problem without sorting. Thus, solving the selection problem by sorting and indexing, as presented in the introduction to this chapter, is asymptotically inefficient.

Exercises

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

9.3-2

Analyze SELECT to show that if $n \geq 140$, then at least $\lceil n/4 \rceil$ elements are greater than the median-of-medians x and at least $\lceil n/4 \rceil$ elements are less than x.

9.3-3

Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

9.3-4 ⋆

Suppose that an algorithm uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i - 1$ smaller elements and the $n - i$ larger elements without performing any additional comparisons.

9.3-5

Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

9.3-6

The kth ***quantiles*** of an n-element set are the $k - 1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

9.3-7

Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

9.3-8

Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

9.3-9

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect

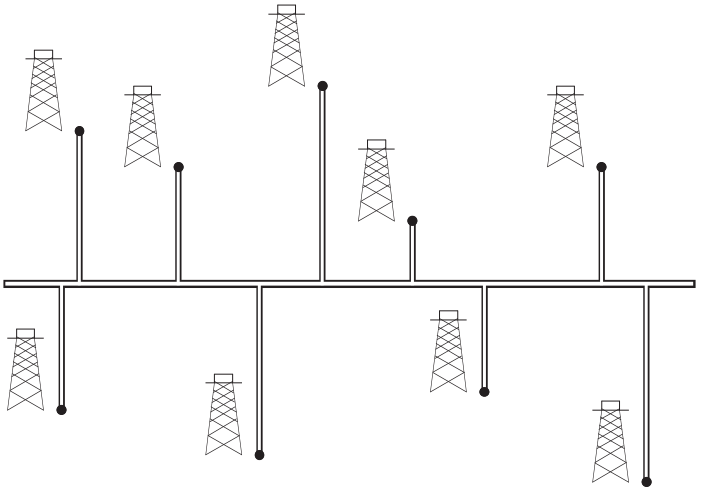


Figure 9.2 Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.2. Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

Problems

9-1 Largest i numbers in sorted order

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

9-2 Weighted median

For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the ***weighted (lower) median*** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} .$$

For example, if the elements are $0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2$ and each element equals its weight (that is, $w_i = x_i$ for $i = 1, 2, \ldots, 7$), then the median is 0.1, but the weighted median is 0.2.

a. Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.

b. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

c. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

The ***post-office location problem*** is defined as follows. We are given n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i \, d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

e. Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-3 Small order statistics

We showed that the worst-case number $T(n)$ of comparisons used by SELECT to select the ith order statistic from n numbers satisfies $T(n) = \Theta(n)$, but the constant hidden by the Θ-notation is rather large. When i is small relative to n, we can implement a different procedure that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

a. Describe an algorithm that uses $U_i(n)$ comparisons to find the ith smallest of n elements, where

$$U_i(n) = \begin{cases} T(n) & \text{if } i \geq n/2\,, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{otherwise}\,. \end{cases}$$

(*Hint:* Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)

b. Show that, if $i < n/2$, then $U_i(n) = n + O(T(2i)\lg(n/i))$.

c. Show that if i is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

d. Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(T(2n/k)\lg k)$.

9-4 Alternative analysis of randomized selection

In this problem, we use indicator random variables to analyze the RANDOMIZED-SELECT procedure in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as $z_1, z_2, \ldots, z_n$, where z_i is the ith smallest element. Thus, the call RANDOMIZED-SELECT$(A, 1, n, k)$ returns z_k.

For $1 \leq i < j \leq n$, let

$$X_{ijk} = \mathrm{I}\{z_i \text{ is compared with } z_j \text{ sometime during the execution of the algorithm to find } z_k\}\,.$$

a. Give an exact expression for $\mathrm{E}[X_{ijk}]$. (*Hint:* Your expression may have different values, depending on the values of i, j, and k.)

b. Let X_k denote the total number of comparisons between elements of array A when finding z_k. Show that

$$\mathrm{E}[X_k] \leq 2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1} + \sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}\right)\,.$$

c. Show that $\mathrm{E}[X_k] \leq 4n$.

d. Conclude that, assuming all elements of array A are distinct, RANDOMIZED-SELECT runs in expected time $O(n)$.

Chapter notes

The worst-case linear-time median-finding algorithm was devised by Blum, Floyd, Pratt, Rivest, and Tarjan [50]. The fast randomized version is due to Hoare [169]. Floyd and Rivest [108] have developed an improved randomized version that partitions around an element recursively selected from a small sample of the elements.

It is still unknown exactly how many comparisons are needed to determine the median. Bent and John [41] gave a lower bound of $2n$ comparisons for median finding, and Schönhage, Paterson, and Pippenger [302] gave an upper bound of $3n$. Dor and Zwick have improved on both of these bounds. Their upper bound [93] is slightly less than $2.95n$, and their lower bound [94] is $(2 + \epsilon)n$, for a small positive constant ϵ, thereby improving slightly on related work by Dor et al. [92]. Paterson [272] describes some of these results along with other related work.

III Data Structures

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets ***dynamic***. The next five chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several different types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. We call a dynamic set that supports these operations a ***dictionary***. Other algorithms require more complicated operations. For example, min-priority queues, which Chapter 6 introduced in the context of the heap data structure, support the operations of inserting an element into and extracting the smallest element from a set. The best way to implement a dynamic set depends upon the operations that must be supported.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated if we have a pointer to the object. (Section 10.3 discusses the implementation of objects and pointers in programming environments that do not contain them as basic data types.) Some kinds of dynamic sets assume that one of the object's attributes is an identifying ***key***. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain ***satellite data***, which are carried around in other object attributes but are otherwise unused by the set implementation. It may

also have attributes that are manipulated by the set operations; these attributes may contain data or pointers to other objects in the set.

Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic ordering. A total ordering allows us to define the minimum element of the set, for example, or to speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: ***queries***, which simply return information about the set, and ***modifying operations***, which change the set. Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

SEARCH(S, k)
: A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
: A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
: A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

MINIMUM(S)
: A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)
: A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

SUCCESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

In some situations, we can extend the queries SUCCESSOR and PREDECESSOR so that they apply to sets with nondistinct keys. For a set on n keys, the normal presumption is that a call to MINIMUM followed by $n - 1$ calls to SUCCESSOR enumerates the elements in the set in sorted order.

We usually measure the time taken to execute a set operation in terms of the size of the set. For example, Chapter 13 describes a data structure that can support any of the operations listed above on a set of size n in time $O(\lg n)$.

Overview of Part III

Chapters 10–14 describe several data structures that we can use to implement dynamic sets; we shall use many of these later to construct efficient algorithms for a variety of problems. We already saw another important data structure—the heap—in Chapter 6.

Chapter 10 presents the essentials of working with simple data structures such as stacks, queues, linked lists, and rooted trees. It also shows how to implement objects and pointers in programming environments that do not support them as primitives. If you have taken an introductory programming course, then much of this material should be familiar to you.

Chapter 11 introduces hash tables, which support the dictionary operations INSERT, DELETE, and SEARCH. In the worst case, hashing requires $\Theta(n)$ time to perform a SEARCH operation, but the expected time for hash-table operations is $O(1)$. The analysis of hashing relies on probability, but most of the chapter requires no background in the subject.

Binary search trees, which are covered in Chapter 12, support all the dynamic-set operations listed above. In the worst case, each operation takes $\Theta(n)$ time on a tree with n elements, but on a randomly built binary search tree, the expected time for each operation is $O(\lg n)$. Binary search trees serve as the basis for many other data structures.

Chapter 13 introduces red-black trees, which are a variant of binary search trees. Unlike ordinary binary search trees, red-black trees are guaranteed to perform well: operations take $O(\lg n)$ time in the worst case. A red-black tree is a balanced search tree; Chapter 18 in Part V presents another kind of balanced search tree, called a B-tree. Although the mechanics of red-black trees are somewhat intricate, you can glean most of their properties from the chapter without studying the mechanics in detail. Nevertheless, you probably will find walking through the code to be quite instructive.

In Chapter 14, we show how to augment red-black trees to support operations other than the basic ones listed above. First, we augment them so that we can dynamically maintain order statistics for a set of keys. Then, we augment them in a different way to maintain intervals of real numbers.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although we can construct many complex data structures using pointers, we present only the rudimentary ones: stacks, queues, linked lists, and rooted trees. We also show ways to synthesize objects and pointers from arrays.

10.1 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. There are several efficient ways to implement stacks and queues on a computer. In this section we show how to use a simple array to implement each.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

As Figure 10.1 shows, we can implement a stack of at most n elements with an array $S[1..n]$. The array has an attribute $S.top$ that indexes the most recently

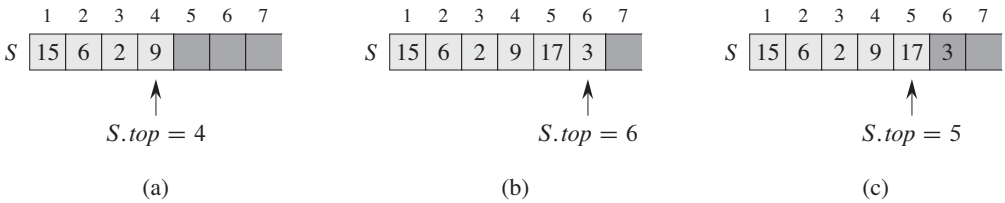


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

inserted element. The stack consists of elements $S[1\,..\,S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.

When $S.top = 0$, the stack contains no elements and is ***empty***. We can test to see whether the stack is empty by query operation STACK-EMPTY. If we attempt to pop an empty stack, we say the stack ***underflows***, which is normally an error. If $S.top$ exceeds n, the stack ***overflows***. (In our pseudocode implementation, we don't worry about stack overflow.)

We can implement each of the stack operations with just a few lines of code:

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

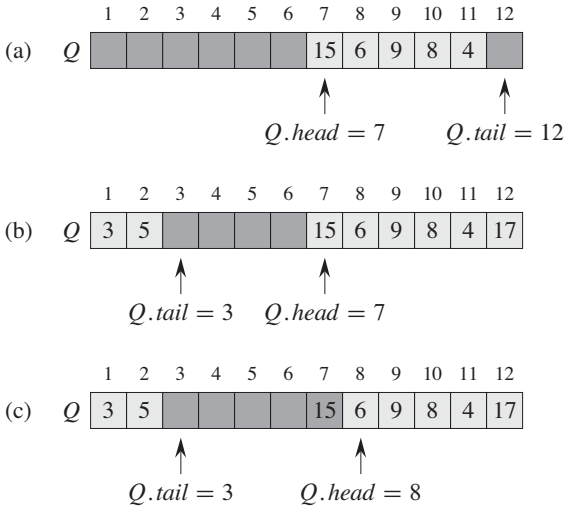


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.

Figure 10.2 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1..n]$. The queue has an attribute $Q.head$ that indexes, or points to, its head. The attribute $Q.tail$ indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue reside in locations $Q.head, Q.head + 1, \ldots, Q.tail - 1$, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order. When $Q.head = Q.tail$, the queue is empty. Initially, we have $Q.head = Q.tail = 1$. If we attempt to dequeue an element from an empty queue, the queue underflows.

When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element, then the queue overflows.

In our procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow. (Exercise 10.1-4 asks you to supply code that checks for these two error conditions.) The pseudocode assumes that $n = Q.length$.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

Exercises

10.1-1

Using Figure 10.1 as a model, illustrate the result of each operation in the sequence PUSH$(S, 4)$, PUSH$(S, 1)$, PUSH$(S, 3)$, POP(S), PUSH$(S, 8)$, and POP(S) on an initially empty stack S stored in array $S[1 \,..\, 6]$.

10.1-2

Explain how to implement two stacks in one array $A[1 \,..\, n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in $O(1)$ time.

10.1-3

Using Figure 10.2 as a model, illustrate the result of each operation in the sequence ENQUEUE$(Q, 4)$, ENQUEUE$(Q, 1)$, ENQUEUE$(Q, 3)$, DEQUEUE(Q), ENQUEUE$(Q, 8)$, and DEQUEUE(Q) on an initially empty queue Q stored in array $Q[1 \,..\, 6]$.

10.1-4

Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

10.1-5
Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a ***deque*** (double-ended queue) allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

10.1-6
Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

10.1-7
Show how to implement a stack using two queues. Analyze the running time of the stack operations.

10.2 Linked lists

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 230.

As shown in Figure 10.3, each element of a ***doubly linked list*** L is an object with an attribute *key* and two other pointer attributes: *next* and *prev*. The object may also contain other satellite data. Given an element x in the list, $x.next$ points to its successor in the linked list, and $x.prev$ points to its predecessor. If $x.prev = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $x.next = \text{NIL}$, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $L.head$ points to the first element of the list. If $L.head = \text{NIL}$, the list is empty.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is ***singly linked***, we omit the *prev* pointer in each element. If a list is ***sorted***, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is then the head of the list, and the maximum element is the tail. If the list is ***unsorted***, the elements can appear in any order. In a ***circular list***, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head. We can think of a circular list as a ring of

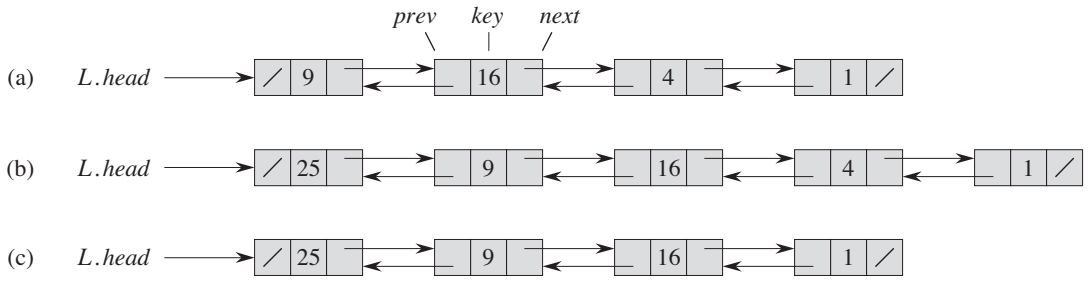


Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

elements. In the remainder of this section, we assume that the lists with which we are working are unsorted and doubly linked.

Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. For the linked list in Figure 10.3(a), the call LIST-SEARCH$(L, 4)$ returns a pointer to the third element, and the call LIST-SEARCH$(L, 7)$ returns NIL.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3       x = x.next
4  return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

Inserting into a linked list

Given an element x whose *key* attribute has already been set, the LIST-INSERT procedure "splices" x onto the front of the linked list, as shown in Figure 10.3(b).

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3       L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

(Recall that our attribute notation can cascade, so that $L.head.prev$ denotes the $prev$ attribute of the object that $L.head$ points to.) The running time for LIST-INSERT on a list of n elements is $O(1)$.

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2       x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5       x.next.prev = x.prev
```

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH to find the element.

Sentinels

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list:

```
LIST-DELETE′(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. For example, suppose that we provide with list L an object $L.nil$ that represents NIL

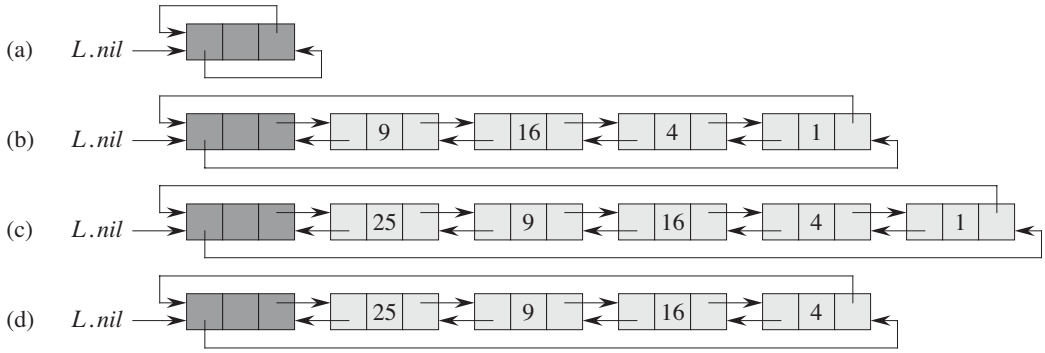


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

but has all the attributes of the other objects in the list. Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel $L.nil$. As shown in Figure 10.4, this change turns a regular doubly linked list into a ***circular, doubly linked list with a sentinel***, in which the sentinel $L.nil$ lies between the head and tail. The attribute $L.nil.next$ points to the head of the list, and $L.nil.prev$ points to the tail. Similarly, both the *next* attribute of the tail and the *prev* attribute of the head point to $L.nil$. Since $L.nil.next$ points to the head, we can eliminate the attribute $L.head$ altogether, replacing references to it by references to $L.nil.next$. Figure 10.4(a) shows that an empty list consists of just the sentinel, and both $L.nil.next$ and $L.nil.prev$ point to $L.nil$.

The code for LIST-SEARCH remains the same as before, but with the references to NIL and $L.head$ changed as specified above:

```
LIST-SEARCH'(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3       x = x.next
4  return x
```

We use the two-line procedure LIST-DELETE$'$ from before to delete an element from the list. The following procedure inserts an element into the list:

```
LIST-INSERT′(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

Figure 10.4 shows the effects of LIST-INSERT′ and LIST-DELETE′ on a sample list.

Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors. The gain from using sentinels within loops is usually a matter of clarity of code rather than speed; the linked list code, for example, becomes simpler when we use sentinels, but we save only $O(1)$ time in the LIST-INSERT′ and LIST-DELETE′ procedures. In other situations, however, the use of sentinels helps to tighten the code in a loop, thus reducing the coefficient of, say, n or n^2 in the running time.

We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. In this book, we use sentinels only when they truly simplify the code.

Exercises

10.2-1

Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?

10.2-2

Implement a stack using a singly linked list L. The operations PUSH and POP should still take $O(1)$ time.

10.2-3

Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

10.2-4

As written, each loop iteration in the LIST-SEARCH′ procedure requires two tests: one for $x \neq L.nil$ and one for $x.key \neq k$. Show how to eliminate the test for $x \neq L.nil$ in each iteration.

10.2-5

Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

10.2-6
The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-7
Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

10.2-8 ⋆
Explain how to implement doubly linked lists using only one pointer value $x.np$ per item instead of the usual two (*next* and *prev*). Assume that all pointer values can be interpreted as k-bit integers, and define $x.np$ to be $x.np = x.next$ XOR $x.prev$, the k-bit "exclusive-or" of $x.next$ and $x.prev$. (The value NIL is represented by 0.) Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

10.3 Implementing pointers and objects

How do we implement pointers and objects in languages that do not provide them? In this section, we shall see two ways of implementing linked data structures without an explicit pointer data type. We shall synthesize objects and pointers from arrays and array indices.

A multiple-array representation of objects

We can represent a collection of objects that have the same attributes by using an array for each attribute. As an example, Figure 10.5 shows how we can implement the linked list of Figure 10.3(a) with three arrays. The array *key* holds the values of the keys currently in the dynamic set, and the pointers reside in the arrays *next* and *prev*. For a given array index x, the array entries $key[x]$, $next[x]$, and $prev[x]$ represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the *key*, *next*, and *prev* arrays.

In Figure 10.3(a), the object with key 4 follows the object with key 16 in the linked list. In Figure 10.5, key 4 appears in $key[2]$, and key 16 appears in $key[5]$, and so $next[5] = 2$ and $prev[2] = 5$. Although the constant NIL appears in the *next*

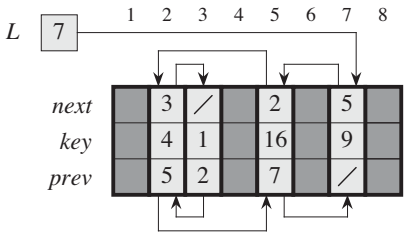


Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays *key*, *next*, and *prev*. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list elements. The variable L keeps the index of the head.

attribute of the tail and the *prev* attribute of the head, we usually use an integer (such as 0 or -1) that cannot possibly represent an actual index into the arrays. A variable L holds the index of the head of the list.

A single-array representation of objects

The words in a computer memory are typically addressed by integers from 0 to $M - 1$, where M is a suitably large integer. In many programming languages, an object occupies a contiguous set of locations in the computer memory. A pointer is simply the address of the first memory location of the object, and we can address other memory locations within the object by adding an offset to the pointer.

We can use the same strategy for implementing objects in programming environments that do not provide explicit pointer data types. For example, Figure 10.6 shows how to use a single array A to store the linked list from Figures 10.3(a) and 10.5. An object occupies a contiguous subarray $A[j\,..\,k]$. Each attribute of the object corresponds to an offset in the range from 0 to $k - j$, and a pointer to the object is the index j. In Figure 10.6, the offsets corresponding to *key*, *next*, and *prev* are 0, 1, and 2, respectively. To read the value of $i.prev$, given a pointer i, we add the value i of the pointer to the offset 2, thus reading $A[i + 2]$.

The single-array representation is flexible in that it permits objects of different lengths to be stored in the same array. The problem of managing such a heterogeneous collection of objects is more difficult than the problem of managing a homogeneous collection, where all objects have the same attributes. Since most of the data structures we shall consider are composed of homogeneous elements, it will be sufficient for our purposes to use the multiple-array representation of objects.

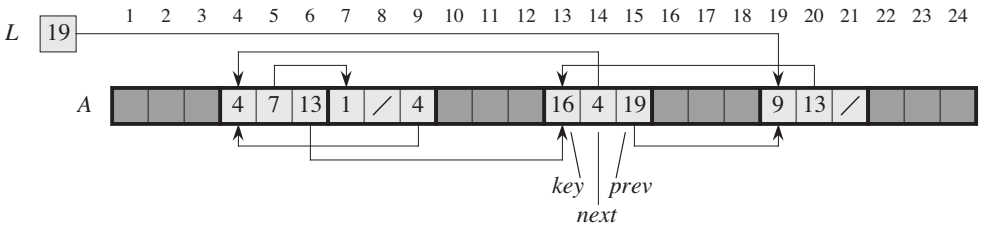


Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array A. Each list element is an object that occupies a contiguous subarray of length 3 within the array. The three attributes *key*, *next*, and *prev* correspond to the offsets 0, 1, and 2, respectively, within each object. A pointer to an object is the index of the first element of the object. Objects containing list elements are lightly shaded, and arrows show the list ordering.

Allocating and freeing objects

To insert a key into a dynamic set represented by a doubly linked list, we must allocate a pointer to a currently unused object in the linked-list representation. Thus, it is useful to manage the storage of objects not currently used in the linked-list representation so that one can be allocated. In some systems, a ***garbage collector*** is responsible for determining which objects are unused. Many applications, however, are simple enough that they can bear responsibility for returning an unused object to a storage manager. We shall now explore the problem of allocating and freeing (or deallocating) homogeneous objects using the example of a doubly linked list represented by multiple arrays.

Suppose that the arrays in the multiple-array representation have length m and that at some moment the dynamic set contains $n \leq m$ elements. Then n objects represent elements currently in the dynamic set, and the remaining $m-n$ objects are ***free***; the free objects are available to represent elements inserted into the dynamic set in the future.

We keep the free objects in a singly linked list, which we call the ***free list***. The free list uses only the *next* array, which stores the *next* pointers within the list. The head of the free list is held in the global variable *free*. When the dynamic set represented by linked list L is nonempty, the free list may be intertwined with list L, as shown in Figure 10.7. Note that each object in the representation is either in list L or in the free list, but not in both.

The free list acts like a stack: the next object allocated is the last one freed. We can use a list implementation of the stack operations PUSH and POP to implement the procedures for allocating and freeing objects, respectively. We assume that the global variable *free* used in the following procedures points to the first element of the free list.

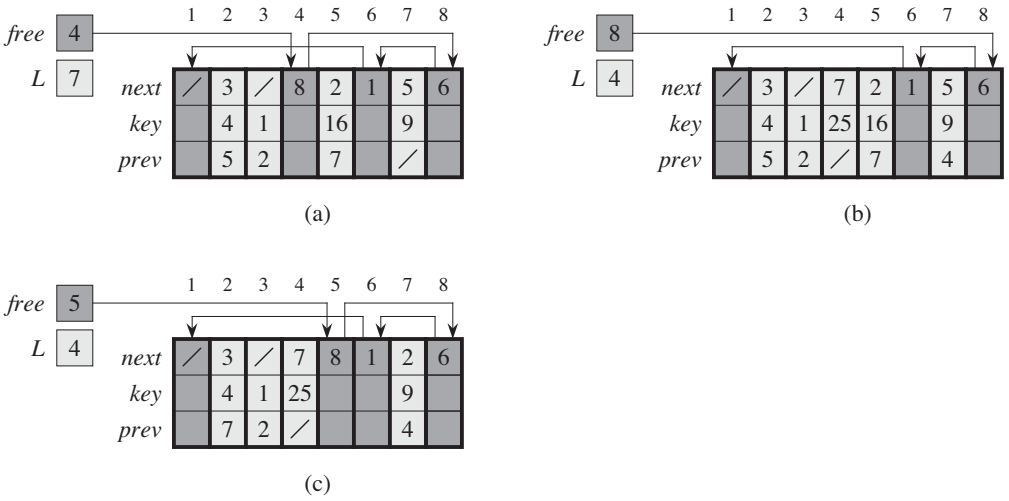


Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. **(a)** The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. **(b)** The result of calling ALLOCATE-OBJECT() (which returns index 4), setting *key*[4] to 25, and calling LIST-INSERT($L, 4$). The new free-list head is object 8, which had been *next*[4] on the free list. **(c)** After executing LIST-DELETE($L, 5$), we call FREE-OBJECT(5). Object 5 becomes the new free-list head, with object 8 following it on the free list.

```
ALLOCATE-OBJECT()
1  if free == NIL
2       error "out of space"
3  else x = free
4       free = x.next
5       return x
```

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

The free list initially contains all n unallocated objects. Once the free list has been exhausted, running the ALLOCATE-OBJECT procedure signals an error. We can even service several linked lists with just a single free list. Figure 10.8 shows two linked lists and a free list intertwined through *key*, *next*, and *prev* arrays.

The two procedures run in $O(1)$ time, which makes them quite practical. We can modify them to work for any homogeneous collection of objects by letting any one of the attributes in the object act like a *next* attribute in the free list.

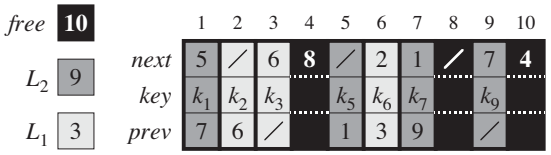


Figure 10.8 Two linked lists, L_1 (lightly shaded) and L_2 (heavily shaded), and a free list (darkened) intertwined.

Exercises

10.3-1

Draw a picture of the sequence $\langle 13, 4, 8, 19, 5, 11 \rangle$ stored as a doubly linked list using the multiple-array representation. Do the same for the single-array representation.

10.3-2

Write the procedures ALLOCATE-OBJECT and FREE-OBJECT for a homogeneous collection of objects implemented by the single-array representation.

10.3-3

Why don't we need to set or reset the *prev* attributes of objects in the implementation of the ALLOCATE-OBJECT and FREE-OBJECT procedures?

10.3-4

It is often desirable to keep all elements of a doubly linked list compact in storage, using, for example, the first m index locations in the multiple-array representation. (This is the case in a paged, virtual-memory computing environment.) Explain how to implement the procedures ALLOCATE-OBJECT and FREE-OBJECT so that the representation is compact. Assume that there are no pointers to elements of the linked list outside the list itself. (*Hint:* Use the array implementation of a stack.)

10.3-5

Let L be a doubly linked list of length n stored in arrays *key*, *prev*, and *next* of length m. Suppose that these arrays are managed by ALLOCATE-OBJECT and FREE-OBJECT procedures that keep a doubly linked free list F. Suppose further that of the m items, exactly n are on list L and $m - n$ are on the free list. Write a procedure COMPACTIFY-LIST(L, F) that, given the list L and the free list F, moves the items in L so that they occupy array positions $1, 2, \ldots, n$ and adjusts the free list F so that it remains correct, occupying array positions $n+1, n+2, \ldots, m$. The running time of your procedure should be $\Theta(n)$, and it should use only a constant amount of extra space. Argue that your procedure is correct.

10.4 Representing rooted trees

The methods for representing lists given in the previous section extend to any homogeneous data structure. In this section, we look specifically at the problem of representing rooted trees by linked data structures. We first look at binary trees, and then we present a method for rooted trees in which nodes can have an arbitrary number of children.

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a *key* attribute. The remaining attributes of interest are pointers to other nodes, and they vary according to the type of tree.

Binary trees

Figure 10.9 shows how we use the attributes p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $x.p = \text{NIL}$, then x is the root. If node x has no left child, then $x.left = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $T.root$. If $T.root = \text{NIL}$, then the tree is empty.

Rooted trees with unbounded branching

We can extend the scheme for representing a binary tree to any class of trees in which the number of children of each node is at most some constant k: we replace the *left* and *right* attributes by $child_1, child_2, \ldots, child_k$. This scheme no longer works when the number of children of a node is unbounded, since we do not know how many attributes (arrays in the multiple-array representation) to allocate in advance. Moreover, even if the number of children k is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme to represent trees with arbitrary numbers of children. It has the advantage of using only $O(n)$ space for any n-node rooted tree. The ***left-child, right-sibling representation*** appears in Figure 10.10. As before, each node contains a parent pointer p, and $T.root$ points to the root of tree T. Instead of having a pointer to each of its children, however, each node x has only two pointers:

1. $x.left\text{-}child$ points to the leftmost child of node x, and
2. $x.right\text{-}sibling$ points to the sibling of x immediately to its right.

If node x has no children, then $x.left\text{-}child = \text{NIL}$, and if node x is the rightmost child of its parent, then $x.right\text{-}sibling = \text{NIL}$.

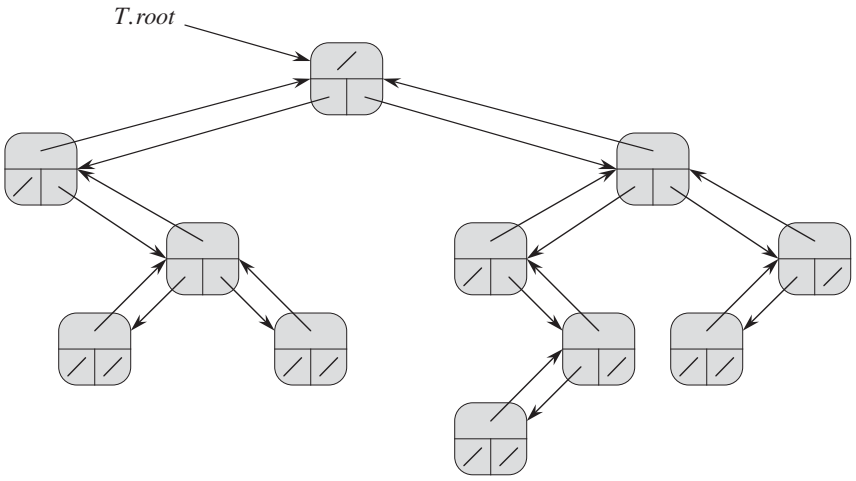


Figure 10.9 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

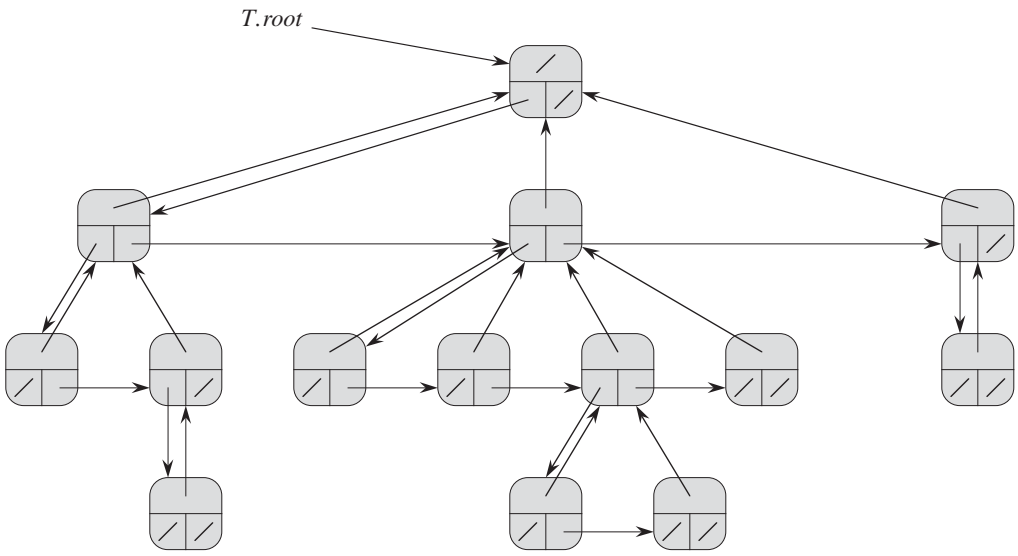


Figure 10.10 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

Other tree representations

We sometimes represent rooted trees in other ways. In Chapter 6, for example, we represented a heap, which is based on a complete binary tree, by a single array plus the index of the last node in the heap. The trees that appear in Chapter 21 are traversed only toward the root, and so only the parent pointers are present; there are no pointers to children. Many other schemes are possible. Which scheme is best depends on the application.

Exercises

10.4-1

Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2

Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

10.4-3

Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

10.4-4

Write an $O(n)$-time procedure that prints all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

10.4-5 ⋆

Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node. Use no more than constant extra space outside

of the tree itself and do not modify the tree, even temporarily, during the procedure.

10.4-6 ⋆

The left-child, right-sibling representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be reached and identified in constant time and all its children can be reached and identified in time linear in the number of children. Show how to use only two pointers and one boolean value in each node so that the parent of a node or all of its children can be reached and identified in time linear in the number of children.

Problems

10-1 Comparisons among lists

For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2 Mergeable heaps using linked lists

A ***mergeable heap*** supports the following operations: MAKE-HEAP (which creates an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION.[1] Show how to implement mergeable heaps using linked lists in each of the following cases. Try to make each operation as efficient as possible. Analyze the running time of each operation in terms of the size of the dynamic set(s) being operated on.

a. Lists are sorted.

b. Lists are unsorted.

c. Lists are unsorted, and dynamic sets to be merged are disjoint.

10-3 Searching a sorted compact list

Exercise 10.3-4 asked how we might maintain an n-element list compactly in the first n positions of an array. We shall assume that all keys are distinct and that the compact list is also sorted, that is, $key[i] < key[next[i]]$ for all $i = 1, 2, \ldots, n$ such that $next[i] \neq \text{NIL}$. We will also assume that we have a variable L that contains the index of the first element on the list. Under these assumptions, you will show that we can use the following randomized algorithm to search the list in $O(\sqrt{n})$ expected time.

```
COMPACT-LIST-SEARCH(L, n, k)
 1  i = L
 2  while i ≠ NIL and key[i] < k
 3        j = RANDOM(1, n)
 4        if key[i] < key[j] and key[j] ≤ k
 5              i = j
 6              if key[i] == k
 7                    return i
 8        i = next[i]
 9  if i == NIL or key[i] > k
10        return NIL
11  else return i
```

If we ignore lines 3–7 of the procedure, we have an ordinary algorithm for searching a sorted linked list, in which index i points to each position of the list in

[1] Because we have defined a mergeable heap to support MINIMUM and EXTRACT-MIN, we can also refer to it as a ***mergeable min-heap***. Alternatively, if it supported MAXIMUM and EXTRACT-MAX, it would be a ***mergeable max-heap***.

turn. The search terminates once the index i "falls off" the end of the list or once $key[i] \geq k$. In the latter case, if $key[i] = k$, clearly we have found a key with the value k. If, however, $key[i] > k$, then we will never find a key with the value k, and so terminating the search was the right thing to do.

Lines 3–7 attempt to skip ahead to a randomly chosen position j. Such a skip benefits us if $key[j]$ is larger than $key[i]$ and no larger than k; in such a case, j marks a position in the list that i would have to reach during an ordinary list search. Because the list is compact, we know that any choice of j between 1 and n indexes some object in the list rather than a slot on the free list.

Instead of analyzing the performance of COMPACT-LIST-SEARCH directly, we shall analyze a related algorithm, COMPACT-LIST-SEARCH$'$, which executes two separate loops. This algorithm takes an additional parameter t which determines an upper bound on the number of iterations of the first loop.

```
COMPACT-LIST-SEARCH'(L, n, k, t)
 1  i = L
 2  for q = 1 to t
 3       j = RANDOM(1, n)
 4       if key[i] < key[j] and key[j] ≤ k
 5            i = j
 6            if key[i] == k
 7                 return i
 8  while i ≠ NIL and key[i] < k
 9       i = next[i]
10  if i == NIL or key[i] > k
11       return NIL
12  else return i
```

To compare the execution of the algorithms COMPACT-LIST-SEARCH(L, n, k) and COMPACT-LIST-SEARCH$'(L, n, k, t)$, assume that the sequence of integers returned by the calls of RANDOM$(1, n)$ is the same for both algorithms.

a. Suppose that COMPACT-LIST-SEARCH(L, n, k) takes t iterations of the **while** loop of lines 2–8. Argue that COMPACT-LIST-SEARCH$'(L, n, k, t)$ returns the same answer and that the total number of iterations of both the **for** and **while** loops within COMPACT-LIST-SEARCH$'$ is at least t.

In the call COMPACT-LIST-SEARCH$'(L, n, k, t)$, let X_t be the random variable that describes the distance in the linked list (that is, through the chain of *next* pointers) from position i to the desired key k after t iterations of the **for** loop of lines 2–7 have occurred.

b. Argue that the expected running time of COMPACT-LIST-SEARCH$'(L, n, k, t)$ is $O(t + \mathrm{E}[X_t])$.

c. Show that $\mathrm{E}[X_t] \le \sum_{r=1}^{n} (1 - r/n)^t$. (*Hint:* Use equation (C.25).)

d. Show that $\sum_{r=0}^{n-1} r^t \le n^{t+1}/(t+1)$.

e. Prove that $\mathrm{E}[X_t] \le n/(t+1)$.

f. Show that COMPACT-LIST-SEARCH$'(L, n, k, t)$ runs in $O(t + n/t)$ expected time.

g. Conclude that COMPACT-LIST-SEARCH runs in $O(\sqrt{n})$ expected time.

h. Why do we assume that all keys are distinct in COMPACT-LIST-SEARCH? Argue that random skips do not necessarily help asymptotically when the list contains repeated key values.

Chapter notes

Aho, Hopcroft, and Ullman [6] and Knuth [209] are excellent references for elementary data structures. Many other texts cover both basic data structures and their implementation in a particular programming language. Examples of these types of textbooks include Goodrich and Tamassia [147], Main [241], Shaffer [311], and Weiss [352, 353, 354]. Gonnet [145] provides experimental data on the performance of many data-structure operations.

The origin of stacks and queues as data structures in computer science is unclear, since corresponding notions already existed in mathematics and paper-based business practices before the introduction of digital computers. Knuth [209] cites A. M. Turing for the development of stacks for subroutine linkage in 1947.

Pointer-based data structures also seem to be a folk invention. According to Knuth, pointers were apparently used in early computers with drum memories. The A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell, J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the use of pointers. Their IPL-III language, developed in 1957, included explicit stack operations.

11 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $O(1)$ time. Section 11.1 discusses direct addressing in more detail. We can take advantage of direct addressing when we can afford to allocate an array that has one position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is *computed* from the key. Section 11.2 presents the main ideas, focusing on "chaining" as a way to handle "collisions," in which more than one key maps to the same array index. Section 11.3 describes how we can compute array indices from keys using hash functions. We present and analyze several variations on the basic theme. Section 11.4 looks at "open addressing," which is another way to deal with collisions. The bottom line is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average. Section 11.5 explains how "perfect hashing" can support searches in $O(1)$ *worst-case* time, when the set of keys being stored is static (that is, when the set of keys never changes once stored).

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large. We shall assume that no two elements have the same key.

To represent the dynamic set, we use an array, or ***direct-address table***, denoted by $T[0..m-1]$, in which each position, or ***slot***, corresponds to a key in the universe U. Figure 11.1 illustrates the approach; slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

The dictionary operations are trivial to implement:

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Each of these operations takes only $O(1)$ time.

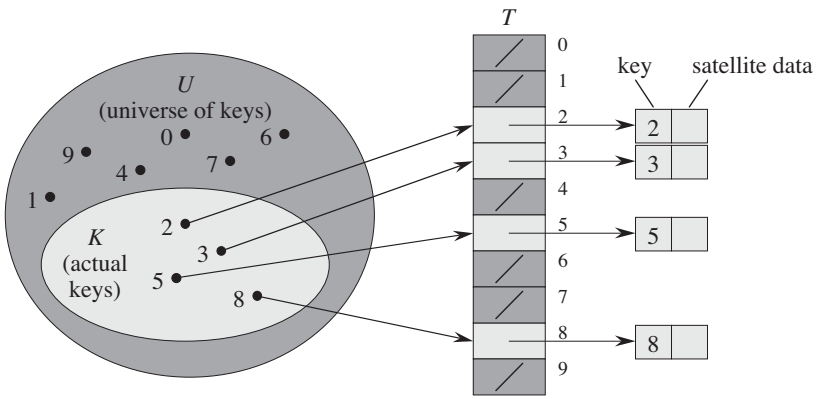


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

For some applications, the direct-address table itself can hold the elements in the dynamic set. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, we can store the object in the slot itself, thus saving space. We would use a special key within an object to indicate an empty slot. Moreover, it is often unnecessary to store the key of the object, since if we have the index of an object in the table, we have its key. If keys are not stored, however, we must have some way to tell whether the slot is empty.

Exercises

11.1-1

Suppose that a dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

11.1-2

A ***bit vector*** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

11.1-3

Suppose how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

11.1-4 ⋆

We wish to implement a dictionary by using direct addressing on a *huge* array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (*Hint:* Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

11.2 Hash tables

The downside of direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, we can reduce the storage requirement to $\Theta(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only $O(1)$ time. The catch is that this bound is for the *average-case time*, whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, we use a ***hash function*** h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a ***hash table*** $T[0\,..\,m-1]$:

$$h : U \to \{0, 1, \ldots, m-1\} \ ,$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k ***hashes*** to slot $h(k)$; we also say that $h(k)$ is the ***hash value*** of key k. Figure 11.2 illustrates the basic idea. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m.

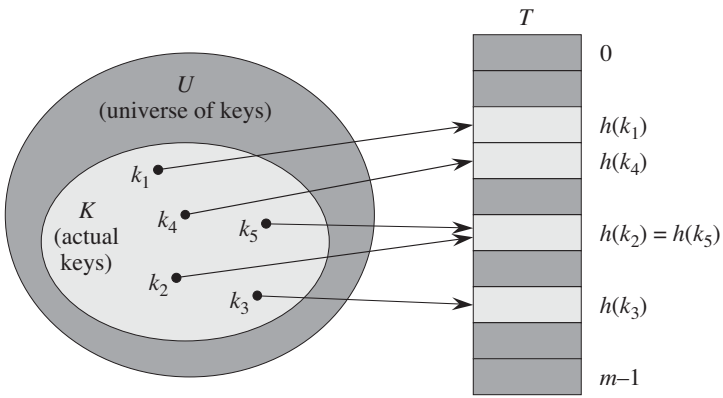


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

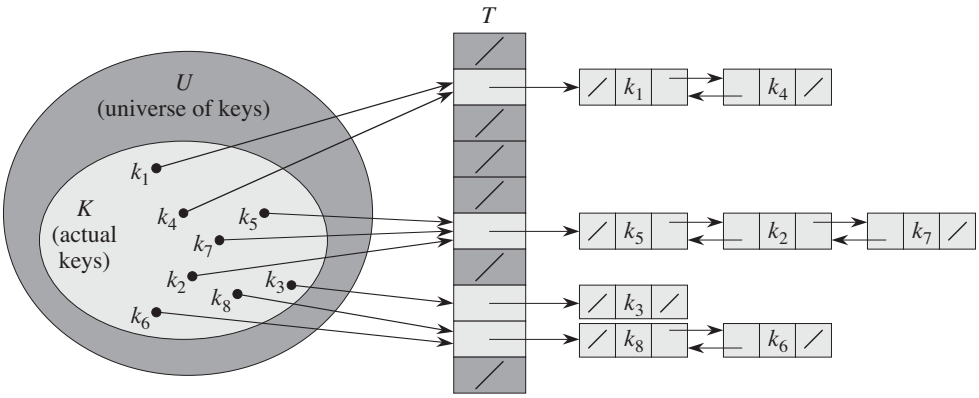


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, we have effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k should always produce the same output $h(k)$.) Because $|U| > m$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible. Thus, while a well-designed, "random"-looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

The remainder of this section presents the simplest collision resolution technique, called chaining. Section 11.4 introduces an alternative method for resolving collisions, called open addressing.

Collision resolution by chaining

In ***chaining***, we place all the elements that hash to the same slot into the same linked list, as Figure 11.3 shows. Slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains NIL.

The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining:

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

The worst-case running time for insertion is $O(1)$. The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table; if necessary, we can check this assumption (at additional cost) by searching for an element whose key is $x.key$ before we insert. For searching, the worst-case running time is proportional to the length of the list; we shall analyze this operation more closely below. We can delete an element in $O(1)$ time if the lists are doubly linked, as Figure 11.3 depicts. (Note that CHAINED-HASH-DELETE takes as input an element x and not its key k, so that we don't have to search for x first. If the hash table supports deletion, then its linked lists should be doubly linked so that we can delete an item quickly. If the lists were only singly linked, then to delete element x, we would first have to find x in the list $T[h(x.key)]$ so that we could update the *next* attribute of x's predecessor. With singly linked lists, both deletion and searching would have the same asymptotic running times.)

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain. Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than if we used one linked list for all the elements. Clearly, we do not use hash tables for their worst-case performance. (Perfect hashing, described in Section 11.5, does provide good worst-case performance when the set of keys is static, however.)

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Section 11.3 discusses these issues, but for now we shall assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of ***simple uniform hashing***.

For $j = 0, 1, \ldots, m-1$, let us denote the length of the list $T[j]$ by n_j, so that

$$n = n_0 + n_1 + \cdots + n_{m-1} , \tag{11.1}$$

and the expected value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.

We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, let us consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k. We shall consider two cases. In the first, the search is unsuccessful: no element in the table has key k. In the second, the search successfully finds an element with key k.

Theorem 11.1
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof Under the assumption of simple uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1 + \alpha)$. ■

The situation for a successful search is slightly different, since each list is not equally likely to be searched. Instead, the probability that a list is searched is proportional to the number of elements it contains. Nonetheless, the expected search time still turns out to be $\Theta(1 + \alpha)$.

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is one more than the number of elements that

appear before x in x's list. Because new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted. To find the expected number of elements examined, we take the average, over the n elements x in the table, of 1 plus the expected number of elements added to x's list after x was added to the list. Let x_i denote the ith element inserted into the table, for $i = 1, 2, \ldots, n$, and let $k_i = x_i.key$. For keys k_i and k_j, we define the indicator random variable $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$. Under the assumption of simple uniform hashing, we have $\Pr\{h(k_i) = h(k_j)\} = 1/m$, and so by Lemma 5.1, $\mathrm{E}[X_{ij}] = 1/m$. Thus, the expected number of elements examined in a successful search is

$$
\begin{aligned}
&\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&\quad = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) \quad \text{(by linearity of expectation)} \\
&\quad = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&\quad = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&\quad = 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&\quad = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{(by equation (A.1))} \\
&\quad = 1+\frac{n-1}{2m} \\
&\quad = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ∎

What does this analysis mean? If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, we can support all dictionary operations in $O(1)$ time on average.

Exercises

11.2-1

Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\} : k \neq l \text{ and } h(k) = h(l)\}$?

11.2-2

Demonstrate what happens when we insert the keys $5, 28, 19, 15, 20, 33, 12, 17, 10$ into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

11.2-5

Suppose that we are storing a set of n keys into a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

11.2-6

Suppose we have stored n keys in a hash table of size m, with collisions resolved by chaining, and that we know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

11.3 Hash functions

In this section, we discuss some issues regarding the design of good hash functions and then present three schemes for their creation. Two of the schemes, hashing by division and hashing by multiplication, are heuristic in nature, whereas the third scheme, universal hashing, uses randomization to provide provably good performance.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally we do know the distribution. For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of simple uniform hashing.

In practice, we can often employ heuristic techniques to create a hash function that performs well. Qualitative information about the distribution of keys may be useful in this design process. For example, consider a compiler's symbol table, in which the keys are character strings representing identifiers in a program. Closely related symbols, such as `pt` and `pts`, often occur in the same program. A good hash function would minimize the chance that such variants hash to the same slot.

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data. For example, the "division method" (discussed in Section 11.3.1) computes the hash value as the remainder when the key is divided by a specified prime number. This method frequently gives good results, assuming that we choose a prime number that is unrelated to any patterns in the distribution of keys.

Finally, we note that some applications of hash functions might require stronger properties than are provided by simple uniform hashing. For example, we might want keys that are "close" in some sense to yield hash values that are far apart. (This property is especially desirable when we are using linear probing, defined in Section 11.4.) Universal hashing, described in Section 11.3.3, often provides the desired properties.

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers. For example, we can interpret a character string as an integer expressed in suitable radix notation. Thus, we might interpret the identifier `pt` as the pair of decimal integers $(112, 116)$, since `p` $= 112$ and `t` $= 116$ in the ASCII character set; then, expressed as a radix-128 integer, `pt` becomes $(112 \cdot 128) + 116 = 14452$. In the context of a given application, we can usually devise some such method for interpreting each key as a (possibly large) natural number. In what follows, we assume that the keys are natural numbers.

11.3.1 The division method

In the ***division method*** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \bmod m \ .$$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k. Unless we know that all low-order p-bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key. As Exercise 11.3-3 asks you to show, choosing $m = 2^p - 1$ when k is a character string interpreted in radix 2^p may be a poor choice, because permuting the characters of k does not change its hash value.

A prime not too close to an exact power of 2 is often a good choice for m. For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$. We could choose $m = 701$ because it is a prime near $2000/3$ but not near any power of 2. Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701 \ .$$

11.3.2 The multiplication method

The ***multiplication method*** for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the

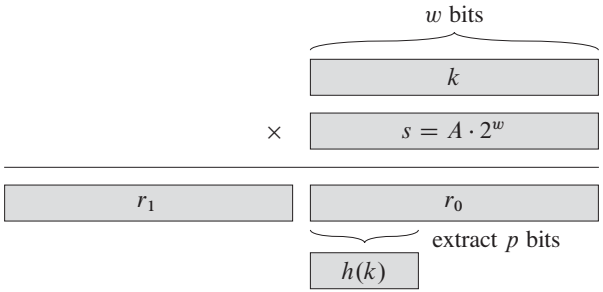


Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor \ ,$$

where "$kA \bmod 1$" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$.

An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), since we can then easily implement the function on most computers as follows. Suppose that the word size of the machine is w bits and that k fits into a single word. We restrict A to be a fraction of the form $s/2^w$, where s is an integer in the range $0 < s < 2^w$. Referring to Figure 11.4, we first multiply k by the w-bit integer $s = A \cdot 2^w$. The result is a $2w$-bit value $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0.

Although this method works with any value of the constant A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Knuth [211] suggests that

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots \tag{11.2}$$

is likely to work reasonably well.

As an example, suppose we have $k = 123456$, $p = 14$, $m = 2^{14} = 16384$, and $w = 32$. Adapting Knuth's suggestion, we choose A to be the fraction of the form $s/2^{32}$ that is closest to $(\sqrt{5} - 1)/2$, so that $A = 2654435769/2^{32}$. Then $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, and so $r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 yield the value $h(k) = 67$.

⋆ 11.3.3 Universal hashing

If a malicious adversary chooses the keys to be hashed by some fixed hash function, then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any fixed hash function is vulnerable to such terrible worst-case behavior; the only effective way to improve the situation is to choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored. This approach, called ***universal hashing***, can yield provably good performance on average, no matter which keys the adversary chooses.

In universal hashing, at the beginning of execution we select the hash function at random from a carefully designed class of functions. As in the case of quicksort, randomization guarantees that no single input will always evoke worst-case behavior. Because we randomly select the hash function, the algorithm can behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input. Returning to the example of a compiler's symbol table, we find that the programmer's choice of identifiers cannot now cause consistently poor hashing performance. Poor performance occurs only when the compiler chooses a random hash function that causes the set of identifiers to hash poorly, but the probability of this situation occurring is small and is the same for any set of identifiers of the same size.

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a collection is said to be ***universal*** if for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys k and l is no more than the chance $1/m$ of a collision if $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m-1\}$.

The following theorem shows that a universal class of hash functions gives good average-case behavior. Recall that n_i denotes the length of list $T[i]$.

Theorem 11.3
Suppose that a hash function h is chosen randomly from a universal collection of hash functions and has been used to hash n keys into a table T of size m, using chaining to resolve collisions. If key k is not in the table, then the expected length $\mathrm{E}[n_{h(k)}]$ of the list that key k hashes to is at most the load factor $\alpha = n/m$. If key k is in the table, then the expected length $\mathrm{E}[n_{h(k)}]$ of the list containing key k is at most $1 + \alpha$.

Proof We note that the expectations here are over the choice of the hash function and do not depend on any assumptions about the distribution of the keys. For each pair k and l of distinct keys, define the indicator random variable

$X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Since by the definition of a universal collection of hash functions, a single pair of keys collides with probability at most $1/m$, we have $\Pr\{h(k) = h(l)\} \le 1/m$. By Lemma 5.1, therefore, we have $\mathrm{E}[X_{kl}] \le 1/m$.

Next we define, for each key k, the random variable Y_k that equals the number of keys other than k that hash to the same slot as k, so that

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl} \ .$$

Thus we have

$$\begin{aligned} \mathrm{E}[Y_k] &= \mathrm{E}\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} \mathrm{E}[X_{kl}] \qquad \text{(by linearity of expectation)} \\ &\le \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m} \ . \end{aligned}$$

The remainder of the proof depends on whether key k is in table T.

- If $k \notin T$, then $n_{h(k)} = Y_k$ and $|\{l : l \in T \text{ and } l \neq k\}| = n$. Thus $\mathrm{E}[n_{h(k)}] = \mathrm{E}[Y_k] \le n/m = \alpha$.
- If $k \in T$, then because key k appears in list $T[h(k)]$ and the count Y_k does not include key k, we have $n_{h(k)} = Y_k + 1$ and $|\{l : l \in T \text{ and } l \neq k\}| = n - 1$. Thus $\mathrm{E}[n_{h(k)}] = \mathrm{E}[Y_k] + 1 \le (n-1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$. ■

The following corollary says universal hashing provides the desired payoff: it has now become impossible for an adversary to pick a sequence of operations that forces the worst-case running time. By cleverly randomizing the choice of hash function at run time, we guarantee that we can process every sequence of operations with a good average-case running time.

Corollary 11.4
Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes expected time $\Theta(n)$ to handle any sequence of n INSERT, SEARCH, and DELETE operations containing $O(m)$ INSERT operations.

Proof Since the number of insertions is $O(m)$, we have $n = O(m)$ and so $\alpha = O(1)$. The INSERT and DELETE operations take constant time and, by Theorem 11.3, the expected time for each SEARCH operation is $O(1)$. By linearity of

expectation, therefore, the expected time for the entire sequence of n operations is $O(n)$. Since each operation takes $\Omega(1)$ time, the $\Theta(n)$ bound follows. ■

Designing a universal class of hash functions

It is quite easy to design a universal class of hash functions, as a little number theory will help us prove. You may wish to consult Chapter 31 first if you are unfamiliar with number theory.

We begin by choosing a prime number p large enough so that every possible key k is in the range 0 to $p-1$, inclusive. Let $\mathbb{Z}_p$ denote the set $\{0, 1, \ldots, p-1\}$, and let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p-1\}$. Since p is prime, we can solve equations modulo p with the methods given in Chapter 31. Because we assume that the size of the universe of keys is greater than the number of slots in the hash table, we have $p > m$.

We now define the hash function h_{ab} for any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$ using a linear transformation followed by reductions modulo p and then modulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \,. \tag{11.3}$$

For example, with $p = 17$ and $m = 6$, we have $h_{3,4}(8) = 5$. The family of all such hash functions is

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\} \,. \tag{11.4}$$

Each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$. This class of hash functions has the nice property that the size m of the output range is arbitrary—not necessarily prime—a feature which we shall use in Section 11.5. Since we have $p-1$ choices for a and p choices for b, the collection $\mathcal{H}_{pm}$ contains $p(p-1)$ hash functions.

Theorem 11.5
The class $\mathcal{H}_{pm}$ of hash functions defined by equations (11.3) and (11.4) is universal.

Proof Consider two distinct keys k and l from $\mathbb{Z}_p$, so that $k \neq l$. For a given hash function h_{ab} we let

$$\begin{aligned} r &= (ak + b) \bmod p \,, \\ s &= (al + b) \bmod p \,. \end{aligned}$$

We first note that $r \neq s$. Why? Observe that

$$r - s \equiv a(k - l) \pmod{p} \,.$$

It follows that $r \neq s$ because p is prime and both a and $(k-l)$ are nonzero modulo p, and so their product must also be nonzero modulo p by Theorem 31.6. Therefore, when computing any $h_{ab} \in \mathcal{H}_{pm}$, distinct inputs k and l map to distinct

values r and s modulo p; there are no collisions yet at the "mod p level." Moreover, each of the possible $p(p-1)$ choices for the pair (a, b) with $a \neq 0$ yields a *different* resulting pair (r, s) with $r \neq s$, since we can solve for a and b given r and s:

$$\begin{aligned} a &= \big((r-s)((k-l)^{-1} \bmod p)\big) \bmod p \,, \\ b &= (r - ak) \bmod p \,, \end{aligned}$$

where $((k-l)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p, of $k-l$. Since there are only $p(p-1)$ possible pairs (r, s) with $r \neq s$, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r, s) with $r \neq s$. Thus, for any given pair of inputs k and l, if we pick (a, b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r, s) is equally likely to be any pair of distinct values modulo p.

Therefore, the probability that distinct keys k and l collide is equal to the probability that $r \equiv s \pmod{m}$ when r and s are randomly chosen as distinct values modulo p. For a given value of r, of the $p-1$ possible remaining values for s, the number of values s such that $s \neq r$ and $s \equiv r \pmod{m}$ is at most

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p+m-1)/m) - 1 \quad \text{(by inequality (3.6))} \\ &= (p-1)/m \,. \end{aligned}$$

The probability that s collides with r when reduced modulo m is at most $((p-1)/m)/(p-1) = 1/m$.

Therefore, for any pair of distinct values $k, l \in \mathbb{Z}_p$,

$$\Pr\{h_{ab}(k) = h_{ab}(l)\} \leq 1/m \,,$$

so that $\mathcal{H}_{pm}$ is indeed universal. ■

Exercises

11.3-1

Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

11.3-2

Suppose that we hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. We can easily represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

11.3-3
Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if we can derive string x from string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

11.3-4
Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m\,(kA \bmod 1)\rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

11.3-5 ⋆
Define a family $\mathcal{H}$ of hash functions from a finite set U to a finite set B to be ***ϵ-universal*** if for all pairs of distinct elements k and l in U,

$$\Pr\{h(k) = h(l)\} \leq \epsilon\ ,$$

where the probability is over the choice of the hash function h drawn at random from the family $\mathcal{H}$. Show that an ϵ-universal family of hash functions must have

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}\ .$$

11.3-6 ⋆
Let U be the set of n-tuples of values drawn from $\mathbb{Z}_p$, and let $B = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \to B$ for $b \in \mathbb{Z}_p$ on an input n-tuple $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ from U as

$$h_b(\langle a_0, a_1, \ldots, a_{n-1}\rangle) = \left(\sum_{j=0}^{n-1} a_j b^j\right) \bmod p\ ,$$

and let $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that $\mathcal{H}$ is $((n-1)/p)$-universal according to the definition of ϵ-universal in Exercise 11.3-5. (*Hint:* See Exercise 31.4-4.)

11.4 Open addressing

In ***open addressing***, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table. No lists and

no elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; one consequence is that the load factor α can never exceed 1.

Of course, we could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \ldots, m-1$ (which requires $\Theta(n)$ search time), the sequence of positions probed *depends upon the key being inserted*. To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\} \ .$$

With open addressing, we require that for every key k, the ***probe sequence***

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. In the following pseudocode, we assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as input a hash table T and a key k. It either returns the slot number where it stores key k or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can

terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. (This argument assumes that keys are not deleted from the hash table.) The procedure HASH-SEARCH takes as input a hash table T and a key k, returning j if it finds that slot j contains key k, or NIL if key k is not present in table T.

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. If we did, we might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupied. We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL. We would then modify the procedure HASH-INSERT to treat such a slot as if it were empty so that we can insert a new key there. We do not need to modify HASH-SEARCH, since it will pass over DELETED values while searching. When we use the special value DELETED, however, search times no longer depend on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

In our analysis, we assume ***uniform hashing***: the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$. Uniform hashing generalizes the notion of simple uniform hashing defined earlier to a hash function that produces not just a single number, but a whole probe sequence. True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

We will examine three commonly used techniques to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all guarantee that $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ for each key k. None of these techniques fulfills the assumption of uniform hashing, however, since none of them is capable of generating more than m^2 different probe sequences (instead of the $m!$ that uniform hashing requires). Double hashing has the greatest number of probe sequences and, as one might expect, seems to give the best results.

Linear probing

Given an ordinary hash function $h' : U \to \{0, 1, \ldots, m-1\}$, which we refer to as an ***auxiliary hash function***, the method of ***linear probing*** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \ldots, m-1$. Given key k, we first probe $T[h'(k)]$, i.e., the slot given by the auxiliary hash function. We next probe slot $T[h'(k)+1]$, and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0], T[1], \ldots$ until we finally probe slot $T[h'(k)-1]$. Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$. Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic probing

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \,, \tag{11.5}$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, \ldots, m-1$. The initial position probed is $T[h'(k)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2, and m are constrained. Problem 11-3 shows one way to select these parameters. Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This property leads to a milder form of clustering, called ***secondary clustering***. As in linear probing, the initial probe determines the entire sequence, and so only m distinct probe sequences are used.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. ***Double hashing*** uses a hash function of the form

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m \,,$$

where both h_1 and h_2 are auxiliary hash functions. The initial probe goes to position $T[h_1(k)]$; successive probe positions are offset from previous positions by the

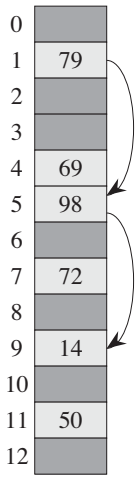


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

amount $h_2(k)$, modulo m. Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary. Figure 11.5 gives an example of insertion by double hashing.

The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. (See Exercise 11.4-4.) A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m. For example, we could choose m prime and let

$$\begin{aligned} h_1(k) &= k \bmod m \ , \\ h_2(k) &= 1 + (k \bmod m') \ , \end{aligned}$$

where m' is chosen to be slightly less than m (say, $m - 1$). For example, if $k = 123456$, $m = 701$, and $m' = 700$, we have $h_1(k) = 80$ and $h_2(k) = 257$, so that we first probe position 80, and then we examine every 257th slot (modulo m) until we find the key or have examined every slot.

When m is prime or a power of 2, double hashing improves over linear or quadratic probing in that $\Theta(m^2)$ probe sequences are used, rather than $\Theta(m)$, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence. As a result, for

such values of m, the performance of double hashing appears to be very close to the performance of the "ideal" scheme of uniform hashing.

Although values of m other than primes or powers of 2 could in principle be used with double hashing, in practice it becomes more difficult to efficiently generate $h_2(k)$ in a way that ensures that it is relatively prime to m, in part because the relative density $\phi(m)/m$ of such numbers may be small (see equation (31.24)).

Analysis of open-address hashing

As in our analysis of chaining, we express our analysis of open addressing in terms of the load factor $\alpha = n/m$ of the hash table. Of course, with open addressing, at most one element occupies each slot, and thus $n \leq m$, which implies $\alpha \leq 1$.

We assume that we are using uniform hashing. In this idealized scheme, the probe sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0, 1, \ldots, m-1\rangle$. Of course, a given key has a unique fixed probe sequence associated with it; what we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.

Theorem 11.6
Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Proof In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty. Let us define the random variable X to be the number of probes made in an unsuccessful search, and let us also define the event A_i, for $i = 1, 2, \ldots$, to be the event that an ith probe occurs and it is to an occupied slot. Then the event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. We will bound $\Pr\{X \geq i\}$ by bounding $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$. By Exercise C.2-5,

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} \ .$$

Since there are n elements and m slots, $\Pr\{A_1\} = n/m$. For $j > 1$, the probability that there is a jth probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-j+1)/(m-j+1)$. This probability follows

because we would be finding one of the remaining $(n - (j - 1))$ elements in one of the $(m - (j - 1))$ unexamined slots, and by the assumption of uniform hashing, the probability is the ratio of these quantities. Observing that $n < m$ implies that $(n - j)/(m - j) \le n/m$ for all j such that $0 \le j < m$, we have for all i such that $1 \le i \le m$,

$$\begin{aligned}\Pr\{X \ge i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\le \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} .\end{aligned}$$

Now, we use equation (C.25) to bound the expected number of probes:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \ge i\} \\ &\le \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} .\end{aligned}$$

■

This bound of $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$ has an intuitive interpretation. We always make the first probe. With probability approximately α, the first probe finds an occupied slot, so that we need to probe a second time. With probability approximately α^2, the first two slots are occupied so that we make a third probe, and so on.

If α is a constant, Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is at most $1/(1 - .5) = 2$. If it is 90 percent full, the average number of probes is at most $1/(1 - .9) = 10$.

Theorem 11.6 gives us the performance of the HASH-INSERT procedure almost immediately.

Corollary 11.7

Inserting an element into an open-address hash table with load factor α requires at most $1/(1 - \alpha)$ probes on average, assuming uniform hashing.

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $1/(1-\alpha)$. ■

We have to do a little more work to compute the expected number of probes for a successful search.

Theorem 11.8
Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} ,$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Proof A search for a key k reproduces the same probe sequence as when the element with key k was inserted. By Corollary 11.7, if k was the $(i+1)$st key inserted into the hash table, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$. Averaging over all n keys in the hash table gives us the expected number of probes in a successful search:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \quad \text{(by inequality (A.12))} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} .
\end{aligned}$$

■

If the hash table is half full, the expected number of probes in a successful search is less than 1.387. If the hash table is 90 percent full, the expected number of probes is less than 2.559.

Exercises

11.4-1
Consider inserting the keys $10, 22, 31, 4, 15, 28, 17, 88, 59$ into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod (m - 1))$.

11.4-2
Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT to handle the special value DELETED.

11.4-3
Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is $3/4$ and when it is $7/8$.

11.4-4 ⋆
Suppose that we use double hashing to resolve collisions—that is, we use the hash function $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Show that if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k, then an unsuccessful search for key k examines $(1/d)$th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table. (*Hint:* See Chapter 31.)

11.4-5 ⋆
Consider an open-address hash table with a load factor α. Find the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by Theorems 11.6 and 11.8 for these expected numbers of probes.

⋆ 11.5 Perfect hashing

Although hashing is often a good choice for its excellent average-case performance, hashing can also provide excellent *worst-case* performance when the set of keys is ***static***: once the keys are stored in the table, the set of keys never changes. Some applications naturally have static sets of keys: consider the set of reserved words in a programming language, or the set of file names on a CD-ROM. We

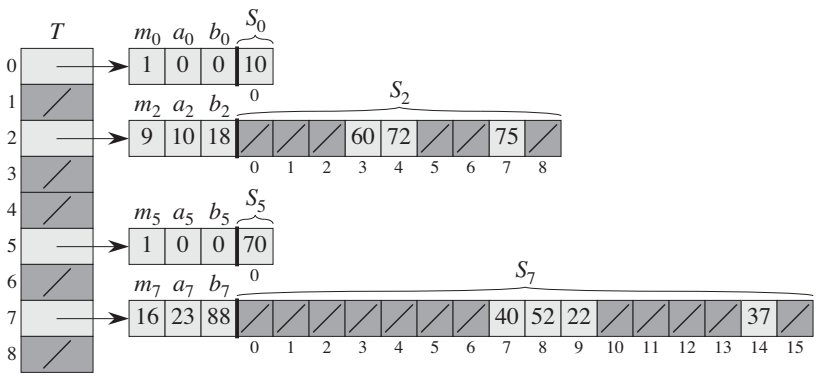


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, and so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is $m_j = n_j^2$, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 7$, key 75 is stored in slot 7 of secondary hash table S_2. No collisions occur in any of the secondary hash tables, and so searching takes constant time in the worst case.

call a hashing technique ***perfect hashing*** if $O(1)$ memory accesses are required to perform a search in the worst case.

To create a perfect hashing scheme, we use two levels of hashing, with universal hashing at each level. Figure 11.6 illustrates the approach.

The first level is essentially the same as for hashing with chaining: we hash the n keys into m slots using a hash function h carefully selected from a family of universal hash functions.

Instead of making a linked list of the keys hashing to slot j, however, we use a small ***secondary hash table*** S_j with an associated hash function h_j. By choosing the hash functions h_j carefully, we can guarantee that there are no collisions at the secondary level.

In order to guarantee that there are no collisions at the secondary level, however, we will need to let the size m_j of hash table S_j be the square of the number n_j of keys hashing to slot j. Although you might think that the quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirement to be excessive, we shall show that by choosing the first-level hash function well, we can limit the expected total amount of space used to $O(n)$.

We use hash functions chosen from the universal classes of hash functions of Section 11.3.3. The first-level hash function comes from the class $\mathcal{H}_{pm}$, where as in Section 11.3.3, p is a prime number greater than any key value. Those keys

hashing to slot j are re-hashed into a secondary hash table S_j of size m_j using a hash function h_j chosen from the class $\mathcal{H}_{p,m_j}$.[1]

We shall proceed in two steps. First, we shall determine how to ensure that the secondary tables have no collisions. Second, we shall show that the expected amount of memory used overall—for the primary hash table and all the secondary hash tables—is $O(n)$.

Theorem 11.9
Suppose that we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions. Then, the probability is less than $1/2$ that there are any collisions.

Proof There are $\binom{n}{2}$ pairs of keys that may collide; each pair collides with probability $1/m$ if h is chosen at random from a universal family $\mathcal{H}$ of hash functions. Let X be a random variable that counts the number of collisions. When $m = n^2$, the expected number of collisions is

$$\begin{aligned} \mathrm{E}[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2 \,. \end{aligned}$$

(This analysis is similar to the analysis of the birthday paradox in Section 5.4.1.) Applying Markov's inequality (C.30), $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$, with $t = 1$, completes the proof. ■

In the situation described in Theorem 11.9, where $m = n^2$, it follows that a hash function h chosen at random from $\mathcal{H}$ is more likely than not to have *no* collisions. Given the set K of n keys to be hashed (remember that K is static), it is thus easy to find a collision-free hash function h with a few random trials.

When n is large, however, a hash table of size $m = n^2$ is excessive. Therefore, we adopt the two-level hashing approach, and we use the approach of Theorem 11.9 only to hash the entries within each slot. We use an outer, or first-level, hash function h to hash the keys into $m = n$ slots. Then, if n_j keys hash to slot j, we use a secondary hash table S_j of size $m_j = n_j^2$ to provide collision-free constant-time lookup.

---

[1] When $n_j = m_j = 1$, we don't really need a hash function for slot j; when we choose a hash function $h_{ab}(k) = ((ak + b) \bmod p) \bmod m_j$ for such a slot, we just use $a = b = 0$.

We now turn to the issue of ensuring that the overall memory used is $O(n)$. Since the size m_j of the jth secondary hash table grows quadratically with the number n_j of keys stored, we run the risk that the overall amount of storage could be excessive.

If the first-level table size is $m = n$, then the amount of memory used is $O(n)$ for the primary hash table, for the storage of the sizes m_j of the secondary hash tables, and for the storage of the parameters a_j and b_j defining the secondary hash functions h_j drawn from the class $\mathcal{H}_{p,m_j}$ of Section 11.3.3 (except when $n_j = 1$ and we use $a = b = 0$). The following theorem and a corollary provide a bound on the expected combined sizes of all the secondary hash tables. A second corollary bounds the probability that the combined size of all the secondary hash tables is superlinear (actually, that it equals or exceeds $4n$).

Theorem 11.10

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions. Then, we have

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n \ ,$$

where n_j is the number of keys hashing to slot j.

Proof We start with the following identity, which holds for any nonnegative integer a:

$$a^2 = a + 2\binom{a}{2} . \tag{11.6}$$

We have

$$\begin{aligned}
&\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \\
&\quad = \mathrm{E}\left[\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(by equation (11.6))} \\
&\quad = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(by linearity of expectation)} \\
&\quad = \mathrm{E}[n] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(by equation (11.1))}
\end{aligned}$$

$$= \quad n + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] \qquad \text{(since } n \text{ is not a random variable)} \ .$$

To evaluate the summation $\sum_{j=0}^{m-1}\binom{n_j}{2}$, we observe that it is just the total number of pairs of keys in the hash table that collide. By the properties of universal hashing, the expected value of this summation is at most

$$\begin{aligned}\binom{n}{2}\frac{1}{m} &= \frac{n(n-1)}{2m} \\ &= \frac{n-1}{2}\ ,\end{aligned}$$

since $m = n$. Thus,

$$\begin{aligned}\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &\le n + 2\,\frac{n-1}{2} \\ &= 2n - 1 \\ &< 2n\ .\end{aligned}$$

■

Corollary 11.11

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \ldots, m-1$. Then, the expected amount of storage required for all secondary hash tables in a perfect hashing scheme is less than $2n$.

Proof Since $m_j = n_j^2$ for $j = 0, 1, \ldots, m-1$, Theorem 11.10 gives

$$\begin{aligned}\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] &= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \\ &< 2n\ ,\end{aligned} \tag{11.7}$$

which completes the proof. ■

Corollary 11.12

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \ldots, m-1$. Then, the probability is less than $1/2$ that the total storage used for secondary hash tables equals or exceeds $4n$.

Proof Again we apply Markov's inequality (C.30), $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$, this time to inequality (11.7), with $X = \sum_{j=0}^{m-1} m_j$ and $t = 4n$:

$$\begin{aligned}\Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} &\leq \frac{\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right]}{4n} \\ &< \frac{2n}{4n} \\ &= 1/2 .\end{aligned}$$

■

From Corollary 11.12, we see that if we test a few randomly chosen hash functions from the universal family, we will quickly find one that uses a reasonable amount of storage.

Exercises

11.5-1 ⋆

Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq e^{-n(n-1)/2m}$. (*Hint:* See equation (3.12).) Argue that when n exceeds $\sqrt{m}$, the probability of avoiding collisions goes rapidly to zero.

Problems

11-1 Longest-probe bound for hashing

Suppose that we use an open-addressed hash table of size m to store $n \leq m/2$ items.

a. Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-k} that the ith insertion requires strictly more than k probes.

b. Show that for $i = 1, 2, \ldots, n$, the probability is $O(1/n^2)$ that the ith insertion requires more than $2 \lg n$ probes.

Let the random variable X_i denote the number of probes required by the ith insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

c. Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.

d. Show that the expected length $\mathrm{E}[X]$ of the longest probe sequence is $O(\lg n)$.

11-2 Slot-size bound for chaining

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg\lg n)$ upper bound on $\mathrm{E}[M]$, the expected value of M.

a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \le n Q_k$.

c. Use Stirling's approximation, equation (3.18), to show that $Q_k < e^k / k^k$.

d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$ for $k_0 = c \lg n / \lg\lg n$. Conclude that $P_k < 1/n^2$ for $k \ge k_0 = c \lg n / \lg\lg n$.

e. Argue that

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c \lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c \lg n}{\lg\lg n}\right\} \cdot \frac{c \lg n}{\lg\lg n} .$$

Conclude that $\mathrm{E}[M] = O(\lg n / \lg\lg n)$.

11-3 Quadratic probing

Suppose that we are given a key k to search for in a hash table with positions $0, 1, \ldots, m-1$, and suppose that we have a hash function h mapping the key space into the set $\{0, 1, \ldots, m-1\}$. The search scheme is as follows:

1. Compute the value $j = h(k)$, and set $i = 0$.
2. Probe in position j for the desired key k. If you find it, or if this position is empty, terminate the search.
3. Set $i = i + 1$. If i now equals m, the table is full, so terminate the search. Otherwise, set $j = (i + j) \bmod m$, and return to step 2.

Assume that m is a power of 2.

a. Show that this scheme is an instance of the general "quadratic probing" scheme by exhibiting the appropriate constants c_1 and c_2 for equation (11.5).

b. Prove that this algorithm examines every table position in the worst case.

11-4 Hashing and authentication

Let $\mathcal{H}$ be a class of hash functions in which each hash function $h \in \mathcal{H}$ maps the universe U of keys to $\{0, 1, \ldots, m-1\}$. We say that $\mathcal{H}$ is ***k*-universal** if, for every fixed sequence of k distinct keys $\langle x^{(1)}, x^{(2)}, \ldots, x^{(k)} \rangle$ and for any h chosen at random from $\mathcal{H}$, the sequence $\langle h(x^{(1)}), h(x^{(2)}), \ldots, h(x^{(k)}) \rangle$ is equally likely to be any of the m^k sequences of length k with elements drawn from $\{0, 1, \ldots, m-1\}$.

a. Show that if the family $\mathcal{H}$ of hash functions is 2-universal, then it is universal.

b. Suppose that the universe U is the set of n-tuples of values drawn from $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is prime. Consider an element $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in U$. For any n-tuple $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle \in U$, define the hash function h_a by

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \bmod p \ .$$

Let $\mathcal{H} = \{h_a\}$. Show that $\mathcal{H}$ is universal, but not 2-universal. (*Hint:* Find a key for which all hash functions in $\mathcal{H}$ produce the same value.)

c. Suppose that we modify $\mathcal{H}$ slightly from part (b): for any $a \in U$ and for any $b \in \mathbb{Z}_p$, define

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

and $\mathcal{H}' = \{h'_{ab}\}$. Argue that $\mathcal{H}'$ is 2-universal. (*Hint:* Consider fixed n-tuples $x \in U$ and $y \in U$, with $x_i \neq y_i$ for some i. What happens to $h'_{ab}(x)$ and $h'_{ab}(y)$ as a_i and b range over $\mathbb{Z}_p$?)

d. Suppose that Alice and Bob secretly agree on a hash function h from a 2-universal family $\mathcal{H}$ of hash functions. Each $h \in \mathcal{H}$ maps from a universe of keys U to $\mathbb{Z}_p$, where p is prime. Later, Alice sends a message m to Bob over the Internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h(m)$, and Bob checks that the pair (m, t) he receives indeed satisfies $t = h(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair (m, t) with a different pair (m', t'). Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has, and even if the adversary knows the family $\mathcal{H}$ of hash functions used.

Chapter notes

Knuth [211] and Gonnet [145] are excellent references for the analysis of hashing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. At about the same time, G. M. Amdahl originated the idea of open addressing.

Carter and Wegman introduced the notion of universal classes of hash functions in 1979 [58].

Fredman, Komlós, and Szemerédi [112] developed the perfect hashing scheme for static sets presented in Section 11.5. An extension of their method to dynamic sets, handling insertions and deletions in amortized expected time $O(1)$, has been given by Dietzfelbinger et al. [86].

12 Binary Search Trees

The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, we can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. We shall see in Section 12.4 that the expected height of a randomly built binary search tree is $O(\lg n)$, so that basic dynamic-set operations on such a tree take $\Theta(\lg n)$ time on average.

In practice, we can't always guarantee that binary search trees are built randomly, but we can design variations of binary search trees with good guaranteed worst-case performance on basic operations. Chapter 13 presents one such variation, red-black trees, which have height $O(\lg n)$. Chapter 18 introduces B-trees, which are particularly good for maintaining databases on secondary (disk) storage.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. We can represent such a tree by a linked data structure in which each node is an object. In addition to a *key* and satellite data, each node contains attributes *left*, *right*, and p that point to the nodes corresponding to its left child,

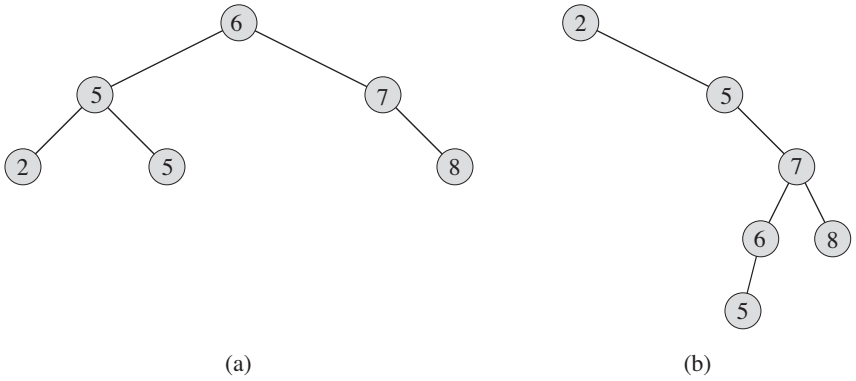


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

Thus, in Figure 12.1(a), the key of the root is 6, the keys 2, 5, and 5 in its left subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller than 6. The same property holds for every node in the tree. For example, the key 5 in the root's left child is no smaller than the key 2 in that node's left subtree and no larger than the key 5 in the right subtree.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.) To use the following procedure to print all the elements in a binary search tree T, we call INORDER-TREE-WALK($T.root$).

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

As an example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order $2, 5, 5, 6, 7, 8$. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1
If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree. Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$. It remains to show that $T(n) = O(n)$.

Since INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq$ NIL), we have $T(0) = c$ for some constant $c > 0$.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is bounded by $T(n) \leq T(k)+T(n-k-1)+d$ for some constant $d > 0$ that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

We use the substitution method to show that $T(n) = O(n)$ by proving that $T(n) \leq (c+d)n+c$. For $n = 0$, we have $(c+d)\cdot 0+c = c = T(0)$. For $n > 0$, we have

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c \ , \end{aligned}$$

which completes the proof. ∎

Exercises

12.1-1
For the set of $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

12.1-2
What is the difference between the binary-search-tree property and the min-heap property (see page 153)? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

12.1-3
Give a nonrecursive algorithm that performs an inorder tree walk. (*Hint:* An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that we can test two pointers for equality.)

12.1-4
Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

12.1-5
Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

We often need to search for a key stored in a binary search tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine these operations and show how to support each one in time $O(h)$ on any binary search tree of height h.

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

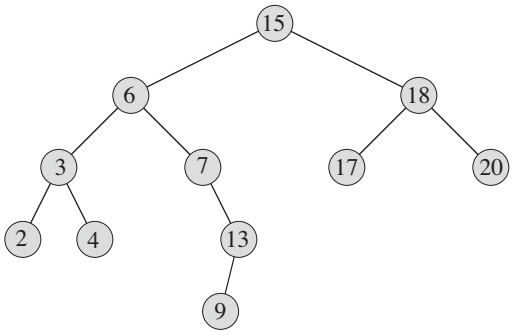


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2       return x
3  if k < x.key
4       return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

The procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure 12.2. For each node x it encounters, it compares the key k with $x.key$. If the two keys are equal, the search terminates. If k is smaller than $x.key$, the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $x.key$, the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

We can rewrite this procedure in an iterative fashion by "unrolling" the recursion into a **while** loop. On most computers, the iterative version is more efficient.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

Minimum and maximum

We can always find an element in a binary search tree whose key is a minimum by following *left* child pointers from the root until we encounter a NIL, as shown in Figure 12.2. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be non-NIL:

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as $x.key$, the minimum key in the subtree rooted at x is $x.key$. If node x has a left subtree, then since no key in the right subtree is smaller than $x.key$ and every key in the left subtree is not larger than $x.key$, the minimum key in the subtree rooted at x resides in the subtree rooted at $x.left$.

The pseudocode for TREE-MAXIMUM is symmetric:

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Successor and predecessor

Given a node in a binary search tree, sometimes we need to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the

successor of a node x is the node with the smallest key greater than $x.key$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree:

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

We break the code for TREE-SUCCESSOR into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree, which we find in line 2 by calling TREE-MINIMUM$(x.right)$. For example, the successor of the node with key 15 in Figure 12.2 is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. In Figure 12.2, the successor of the node with key 13 is the node with key 15. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3–7 of TREE-SUCCESSOR handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a simple path up the tree or follow a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x), respectively.

In summary, we have proved the following theorem.

Theorem 12.2

We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in $O(h)$ time on a binary search tree of height h. ■

Exercises

12.2-1

Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2

Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

12.2-3

Write the TREE-PREDECESSOR procedure.

12.2-4

Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

12.2-5

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2-6

Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

12.2-7

An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

12.2-8
Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

12.2-9
Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

Insertion

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure takes a node z for which $z.key = v$, $z.left =$ NIL, and $z.right =$ NIL. It modifies T and some of the attributes of z in such a way that it inserts z into an appropriate position in the tree.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ 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 == NIL
10       T.root = z          // tree T was empty
11  elseif z.key < y.key
12       y.left = z
13  else y.right = z
```

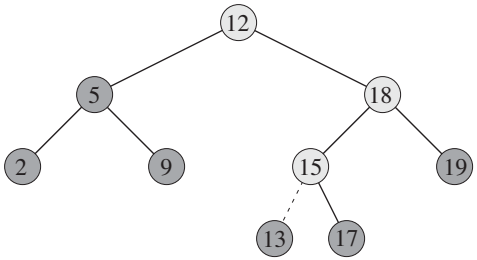


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and the pointer x traces a simple path downward looking for a NIL to replace with the input item z. The procedure maintains the ***trailing pointer*** y as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $z.key$ with $x.key$, until x becomes NIL. This NIL occupies the position where we wish to place the input item z. We need the trailing pointer y, because by the time we find the NIL where z belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases but, as we shall see, one of the cases is a bit tricky.

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, then we find z's successor y—which must be in z's right subtree—and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. This case is the tricky one because, as we shall see, it matters whether y is z's right child.

The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z. It organizes its cases a bit differently from the three cases outlined previously by considering the four cases shown in Figure 12.4.

- If z has no left child (part (a) of the figure), then we replace z by its right child, which may or may not be NIL. When z's right child is NIL, this case deals with the situation in which z has no children. When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.
- If z has just one child, which is its left child (part (b) of the figure), then we replace z by its left child.
- Otherwise, z has both a left and a right child. We find z's successor y, which lies in z's right subtree and has no left child (see Exercise 12.2-5). We want to splice y out of its current location and have it replace z in the tree.
  - If y is z's right child (part (c)), then we replace z by y, leaving y's right child alone.
  - Otherwise, y lies within z's right subtree but is not z's right child (part (d)). In this case, we first replace y by its own right child, and then we replace z by y.

In order to move subtrees around within the binary search tree, we define a subroutine TRANSPLANT, which replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

Lines 1–2 handle the case in which u is the root of T. Otherwise, u is either a left child or a right child of its parent. Lines 3–4 take care of updating $u.p.left$ if u is a left child, and line 5 updates $u.p.right$ if u is a right child. We allow v to be NIL, and lines 6–7 update $v.p$ if v is non-NIL. Note that TRANSPLANT does not attempt to update $v.left$ and $v.right$; doing so, or not doing so, is the responsibility of TRANSPLANT's caller.

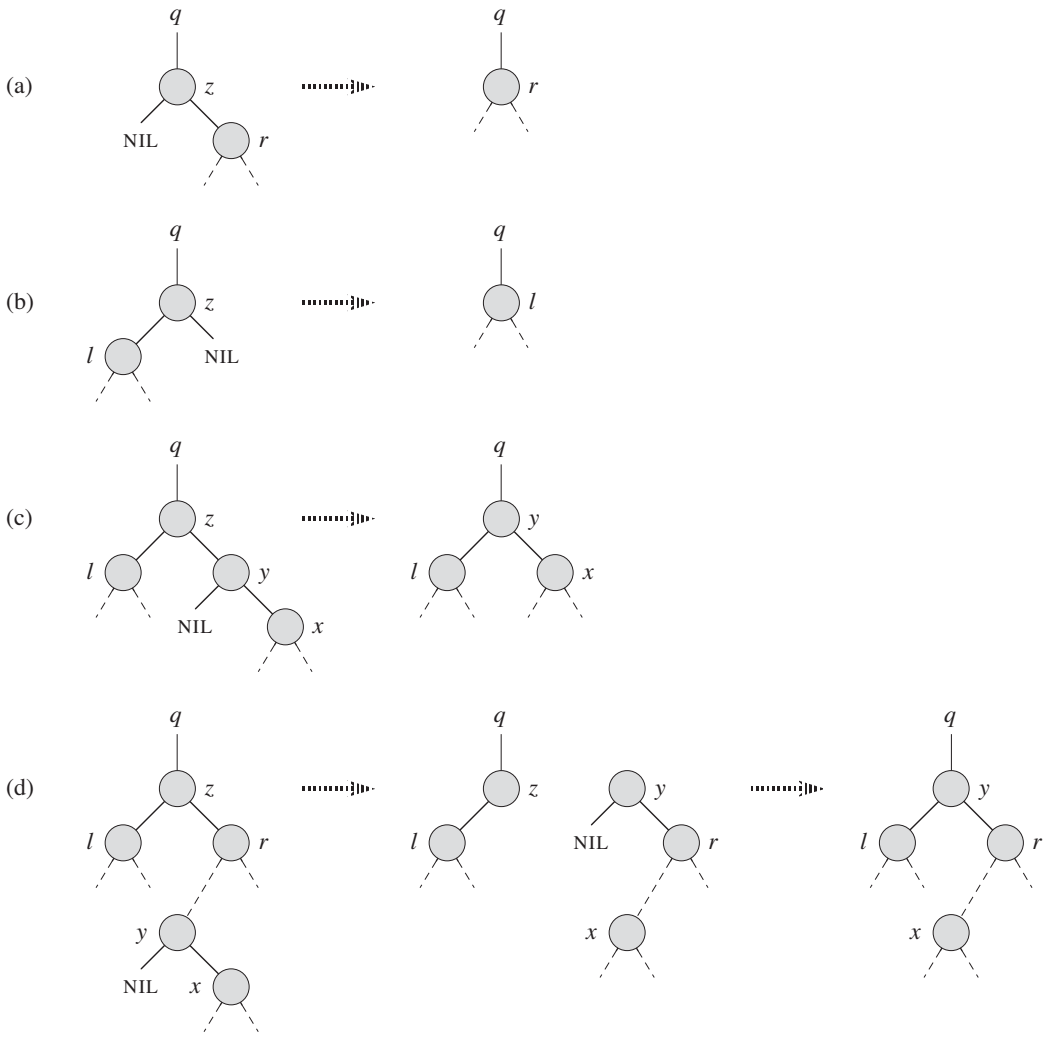


Figure 12.4 Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. **(a)** Node z has no left child. We replace z by its right child r, which may or may not be NIL. **(b)** Node z has a left child l but no right child. We replace z by l. **(c)** Node z has two children; its left child is node l, its right child is its successor y, and y's right child is node x. We replace z by y, updating y's left child to become l, but leaving x as y's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. We replace y by its own right child x, and we set y to be r's parent. Then, we set y to be q's child and the parent of l.

With the TRANSPLANT procedure in hand, here is the procedure that deletes node z from binary search tree T:

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

The TREE-DELETE procedure executes the four cases as follows. Lines 1–2 handle the case in which node z has no left child, and lines 3–4 handle the case in which z has a left child but no right child. Lines 5–12 deal with the remaining two cases, in which z has two children. Line 5 finds node y, which is the successor of z. Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key; hence the call to TREE-MINIMUM$(z.right)$. As we noted before, y has no left child. We want to splice y out of its current location, and it should replace z in the tree. If y is z's right child, then lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child. If y is not z's left child, lines 7–9 replace y as a child of its parent by y's right child and turn z's right child into y's right child, and then lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child.

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs in $O(h)$ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3

We can implement the dynamic-set operations INSERT and DELETE so that each one runs in $O(h)$ time on a binary search tree of height h. ■

Exercises

12.3-1
Give a recursive version of the TREE-INSERT procedure.

12.3-2
Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

12.3-3
We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

12.3-4
Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

12.3-5
Suppose that instead of each node x keeping the attribute $x.p$, pointing to x's parent, it keeps $x.succ$, pointing to x's successor. Give pseudocode for SEARCH, INSERT, and DELETE on a binary search tree T using this representation. These procedures should operate in time $O(h)$, where h is the height of the tree T. (*Hint:* You may wish to implement a subroutine that returns the parent of a node.)

12.3-6
When node z in TREE-DELETE has two children, we could choose node y as its predecessor rather than its successor. What other changes to TREE-DELETE would be necessary if we did so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be changed to implement such a fair strategy?

⋆ 12.4 Randomly built binary search trees

We have shown that each of the basic operations on a binary search tree runs in $O(h)$ time, where h is the height of the tree. The height of a binary search

tree varies, however, as items are inserted and deleted. If, for example, the n items are inserted in strictly increasing order, the tree will be a chain with height $n - 1$. On the other hand, Exercise B.5-4 shows that $h \geq \lfloor \lg n \rfloor$. As with quicksort, we can show that the behavior of the average case is much closer to the best case than to the worst case.

Unfortunately, little is known about the average height of a binary search tree when both insertion and deletion are used to create it. When the tree is created by insertion alone, the analysis becomes more tractable. Let us therefore define a ***randomly built binary search tree*** on n keys as one that arises from inserting the keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely. (Exercise 12.4-3 asks you to show that this notion is different from assuming that every binary search tree on n keys is equally likely.) In this section, we shall prove the following theorem.

Theorem 12.4
The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$.

Proof We start by defining three random variables that help measure the height of a randomly built binary search tree. We denote the height of a randomly built binary search on n keys by X_n, and we define the ***exponential height*** $Y_n = 2^{X_n}$. When we build a binary search tree on n keys, we choose one key as that of the root, and we let R_n denote the random variable that holds this key's ***rank*** within the set of n keys; that is, R_n holds the position that this key would occupy if the set of keys were sorted. The value of R_n is equally likely to be any element of the set $\{1, 2, \ldots, n\}$. If $R_n = i$, then the left subtree of the root is a randomly built binary search tree on $i - 1$ keys, and the right subtree is a randomly built binary search tree on $n - i$ keys. Because the height of a binary tree is 1 more than the larger of the heights of the two subtrees of the root, the exponential height of a binary tree is twice the larger of the exponential heights of the two subtrees of the root. If we know that $R_n = i$, it follows that

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) \ .$$

As base cases, we have that $Y_1 = 1$, because the exponential height of a tree with 1 node is $2^0 = 1$ and, for convenience, we define $Y_0 = 0$.

Next, define indicator random variables $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, where

$$Z_{n,i} = \mathrm{I}\{R_n = i\} \ .$$

Because R_n is equally likely to be any element of $\{1, 2, \ldots, n\}$, it follows that $\Pr\{R_n = i\} = 1/n$ for $i = 1, 2, \ldots, n$, and hence, by Lemma 5.1, we have

$$\mathrm{E}[Z_{n,i}] = 1/n \ , \tag{12.1}$$

for $i = 1, 2, \ldots, n$. Because exactly one value of $Z_{n,i}$ is 1 and all others are 0, we also have

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right) .$$

We shall show that $\mathrm{E}[Y_n]$ is polynomial in n, which will ultimately imply that $\mathrm{E}[X_n] = O(\lg n)$.

We claim that the indicator random variable $Z_{n,i} = \mathrm{I}\{R_n = i\}$ is independent of the values of Y_{i-1} and Y_{n-i}. Having chosen $R_n = i$, the left subtree (whose exponential height is Y_{i-1}) is randomly built on the $i - 1$ keys whose ranks are less than i. This subtree is just like any other randomly built binary search tree on $i - 1$ keys. Other than the number of keys it contains, this subtree's structure is not affected at all by the choice of $R_n = i$, and hence the random variables Y_{i-1} and $Z_{n,i}$ are independent. Likewise, the right subtree, whose exponential height is Y_{n-i}, is randomly built on the $n - i$ keys whose ranks are greater than i. Its structure is independent of the value of R_n, and so the random variables Y_{n-i} and $Z_{n,i}$ are independent. Hence, we have

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\left[Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\, \mathrm{E}\left[2 \cdot \max(Y_{i-1}, Y_{n-i})\right] && \text{(by independence)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[2 \cdot \max(Y_{i-1}, Y_{n-i})\right] && \text{(by equation (12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}\left[\max(Y_{i-1}, Y_{n-i})\right] && \text{(by equation (C.22))} \\
&\le \frac{2}{n} \sum_{i=1}^{n} \left(\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]\right) && \text{(by Exercise C.3-4)} .
\end{aligned}$$

Since each term $\mathrm{E}[Y_0], \mathrm{E}[Y_1], \ldots, \mathrm{E}[Y_{n-1}]$ appears twice in the last summation, once as $\mathrm{E}[Y_{i-1}]$ and once as $\mathrm{E}[Y_{n-i}]$, we have the recurrence

$$\mathrm{E}[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i] . \tag{12.2}$$

Using the substitution method, we shall show that for all positive integers n, the recurrence (12.2) has the solution

$$\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3}.$$

In doing so, we shall use the identity

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4}. \tag{12.3}$$

(Exercise 12.4-1 asks you to prove this identity.)

For the base cases, we note that the bounds $0 = Y_0 = \mathrm{E}[Y_0] \le (1/4)\binom{3}{3} = 1/4$ and $1 = Y_1 = \mathrm{E}[Y_1] \le (1/4)\binom{1+3}{3} = 1$ hold. For the inductive case, we have that

$$\begin{aligned}
\mathrm{E}[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \\
&\le \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(by the inductive hypothesis)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(by equation (12.3))} \\
&= \frac{1}{n}\cdot\frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4}\cdot\frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3}.
\end{aligned}$$

We have bounded $\mathrm{E}[Y_n]$, but our ultimate goal is to bound $\mathrm{E}[X_n]$. As Exercise 12.4-4 asks you to show, the function $f(x) = 2^x$ is convex (see page 1199). Therefore, we can employ Jensen's inequality (C.26), which says that

$$\begin{aligned}
2^{\mathrm{E}[X_n]} &\le \mathrm{E}\left[2^{X_n}\right] \\
&= \mathrm{E}[Y_n]\ ,
\end{aligned}$$

as follows:

$$2^{\mathrm{E}[X_n]} \le \frac{1}{4}\binom{n+3}{3}$$

$$= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6}$$
$$= \frac{n^3 + 6n^2 + 11n + 6}{24} .$$

Taking logarithms of both sides gives $\mathrm{E}[X_n] = O(\lg n)$. ■

Exercises

12.4-1
Prove equation (12.3).

12.4-2
Describe a binary search tree on n nodes such that the average depth of a node in the tree is $\Theta(\lg n)$ but the height of the tree is $\omega(\lg n)$. Give an asymptotic upper bound on the height of an n-node binary search tree in which the average depth of a node is $\Theta(\lg n)$.

12.4-3
Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (*Hint:* List the possibilities when $n = 3$.)

12.4-4
Show that the function $f(x) = 2^x$ is convex.

12.4-5 ⋆
Consider RANDOMIZED-QUICKSORT operating on a sequence of n distinct input numbers. Prove that for any constant $k > 0$, all but $O(1/n^k)$ of the $n!$ input permutations yield an $O(n \lg n)$ running time.

Problems

12-1 Binary search trees with equal keys
Equal keys pose a problem for the implementation of binary search trees.

a. What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

We propose to improve TREE-INSERT by testing before line 5 to determine whether $z.key = x.key$ and by testing before line 11 to determine whether $z.key = y.key$.

If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which we compare the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)

b. Keep a boolean flag $x.b$ at node x, and set x to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time we visit x while inserting a node with the same key as x.

c. Keep a list of nodes with equal keys at x, and insert z into the list.

d. Randomly set x to either $x.left$ or $x.right$. (Give the worst-case performance and informally derive the expected running time.)

12-2 Radix trees

Given two strings $a = a_0a_1 \ldots a_p$ and $b = b_0b_1 \ldots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either

1. there exists an integer j, where $0 \le j \le \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \ldots, j - 1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.

The ***radix tree*** data structure shown in Figure 12.5 stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0a_1 \ldots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

12-3 Average node depth in a randomly built binary search tree

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. Although this result is weaker than that of Theorem 12.4, the technique we shall use reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.

We define the ***total path length*** $P(T)$ of a binary tree T as the sum, over all nodes x in T, of the depth of node x, which we denote by $d(x, T)$.

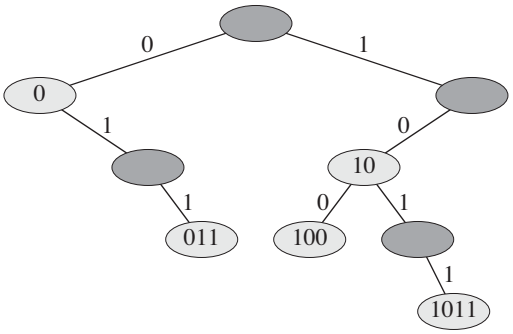


Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. We can determine each node's key by traversing the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes; the keys appear here for illustrative purposes only. Nodes are heavily shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish a path to other nodes.

a. Argue that the average depth of a node in T is

$$\frac{1}{n}\sum_{x \in T} d(x, T) = \frac{1}{n} P(T) \ .$$

Thus, we wish to show that the expected value of $P(T)$ is $O(n \lg n)$.

b. Let T_L and T_R denote the left and right subtrees of tree T, respectively. Argue that if T has n nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1 \ .$$

c. Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n - i - 1) + n - 1) \ .$$

d. Show how to rewrite $P(n)$ as

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) \ .$$

e. Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$.

At each recursive invocation of quicksort, we choose a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

f. Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

12-4 Number of different binary trees

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n, as well as an asymptotic estimate.

a. Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} \ .$$

b. Referring to Problem 4-4 for the definition of a generating function, let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n \ .$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1 - 4x}\right) \ .$$

The ***Taylor expansion*** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x - a)^k \ ,$$

where $f^{(k)}(x)$ is the kth derivative of f evaluated at x.

c. Show that

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(the nth ***Catalan number***) by using the Taylor expansion of $\sqrt{1-4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial expansion (C.4) to nonintegral exponents n, where for any real number n and for any integer k, we interpret $\binom{n}{k}$ to be $n(n-1)\cdots(n-k+1)/k!$ if $k \geq 0$, and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}\left(1 + O(1/n)\right) .$$

Chapter notes

Knuth [211] contains a good discussion of simple binary search trees as well as many variations. Binary search trees seem to have been independently discovered by a number of people in the late 1950s. Radix trees are often called "tries," which comes from the middle letters in the word *retrieval*. Knuth [211] also discusses them.

Many texts, including the first two editions of this book, have a somewhat simpler method of deleting a node from a binary search tree when both of its children are present. Instead of replacing node z by its successor y, we delete node y but copy its key and satellite data into node z. The downside of this approach is that the node actually deleted might not be the node passed to the delete procedure. If other components of a program maintain pointers to nodes in the tree, they could mistakenly end up with "stale" pointers to nodes that have been deleted. Although the deletion method presented in this edition of this book is a bit more complicated, it guarantees that a call to delete node z deletes node z and only node z.

Section 15.5 will show how to construct an optimal binary search tree when we know the search frequencies before constructing the tree. That is, given the frequencies of searching for each key and the frequencies of searching for values that fall between keys in the tree, we construct a binary search tree for which a set of searches that follows these frequencies examines the minimum number of nodes.

The proof in Section 12.4 that bounds the expected height of a randomly built binary search tree is due to Aslam [24]. Martínez and Roura [243] give randomized algorithms for insertion into and deletion from binary search trees in which the result of either operation is a random binary search tree. Their definition of a random binary search tree differs—only slightly—from that of a randomly built binary search tree in this chapter, however.

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 $\mathrm{bh}(x)$. 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 $2^{\mathrm{bh}(x)} - 1$ 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 $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ 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 $\mathrm{bh}(x)$ or $\mathrm{bh}(x) - 1$, 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 $2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ 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

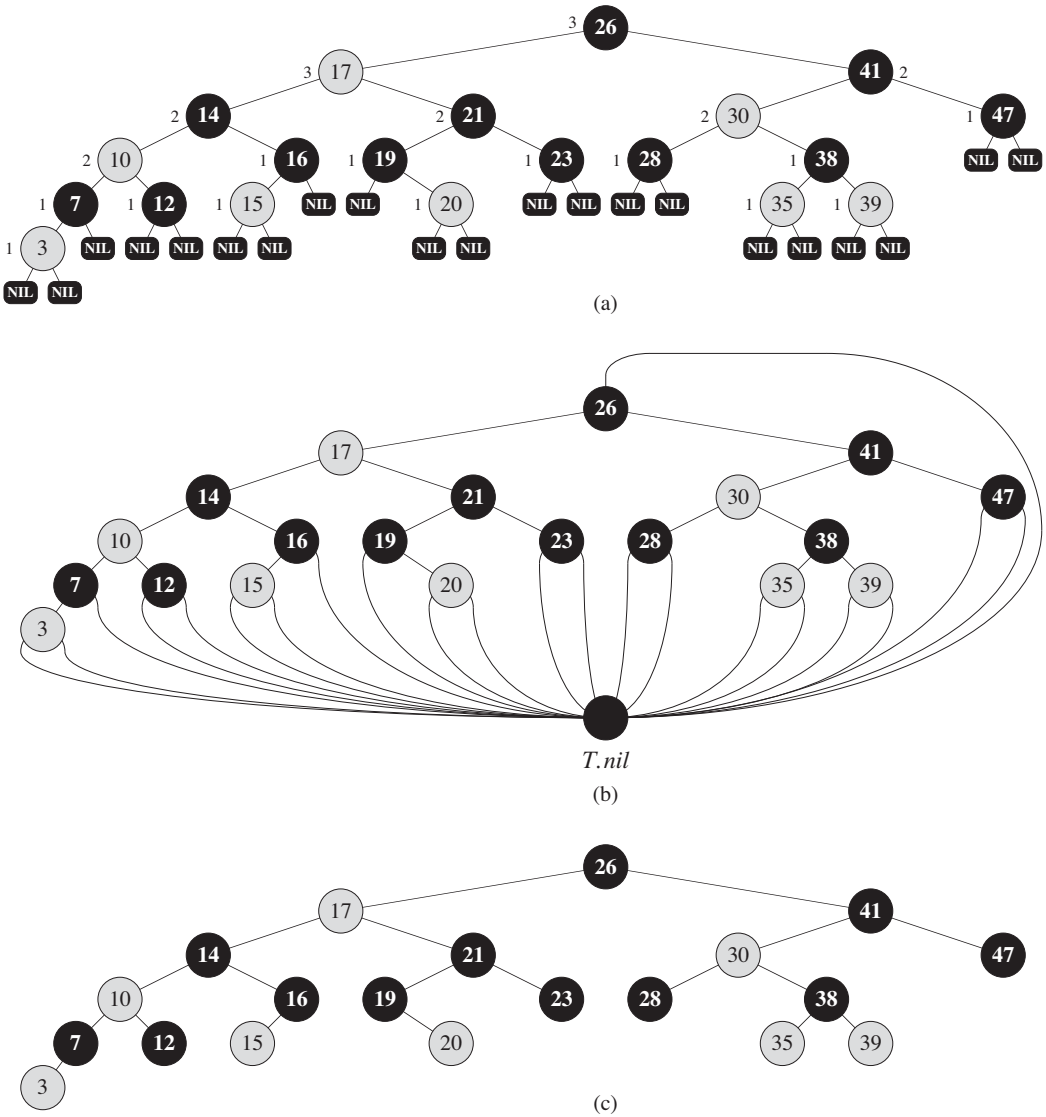


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a 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 $T.nil$.

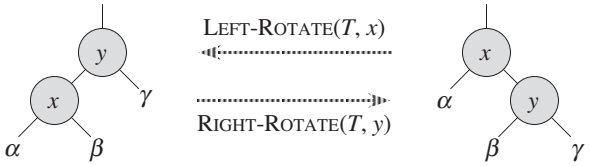


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on 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.

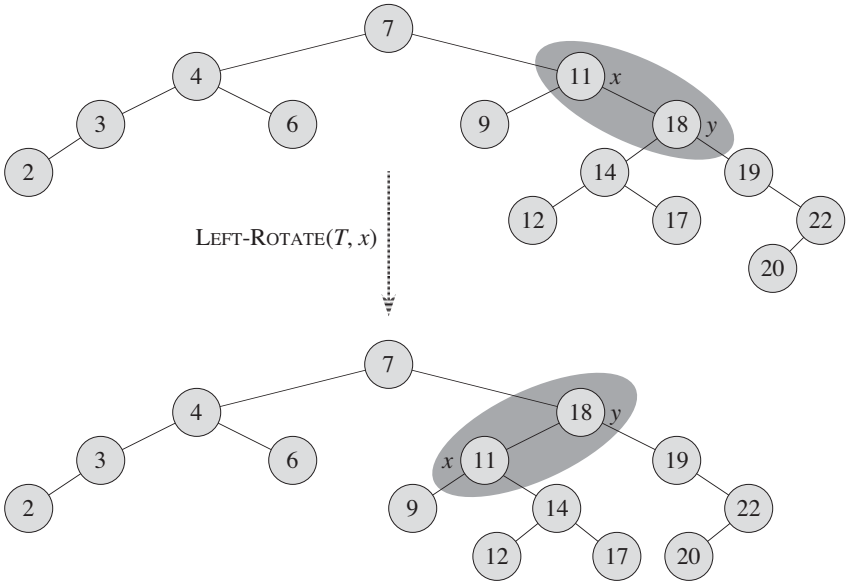


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

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 ≠ 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.

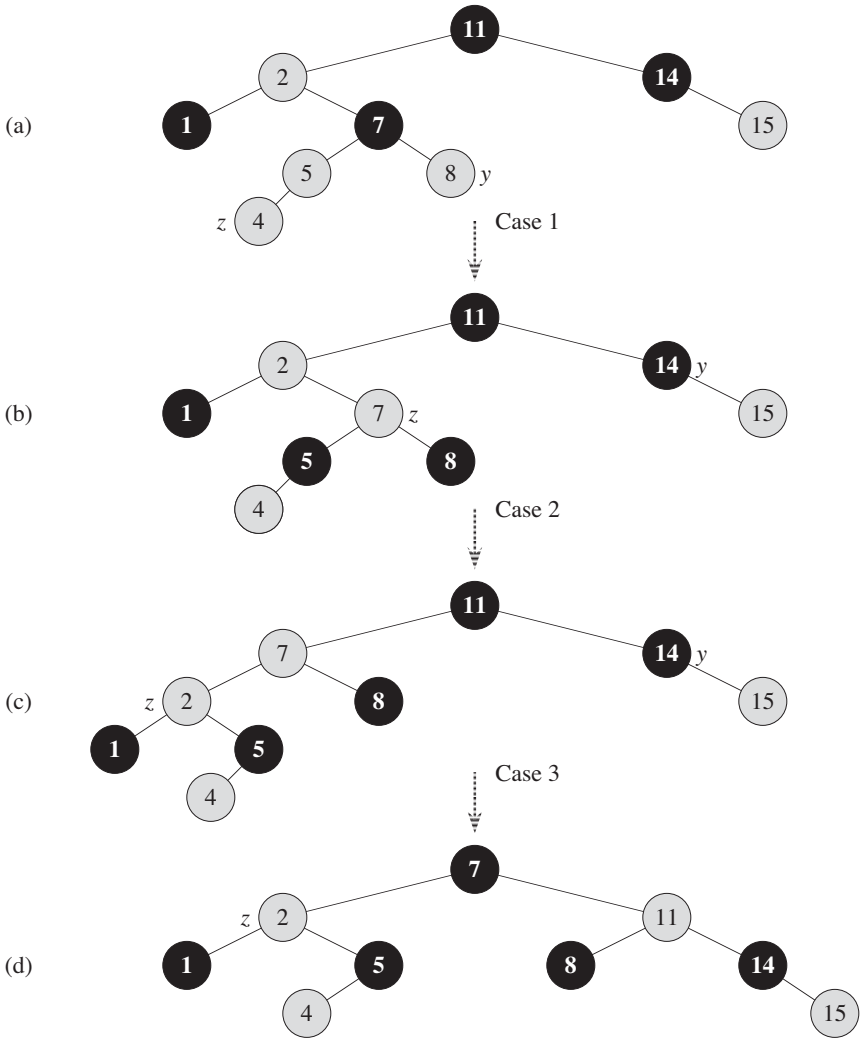


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. 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.

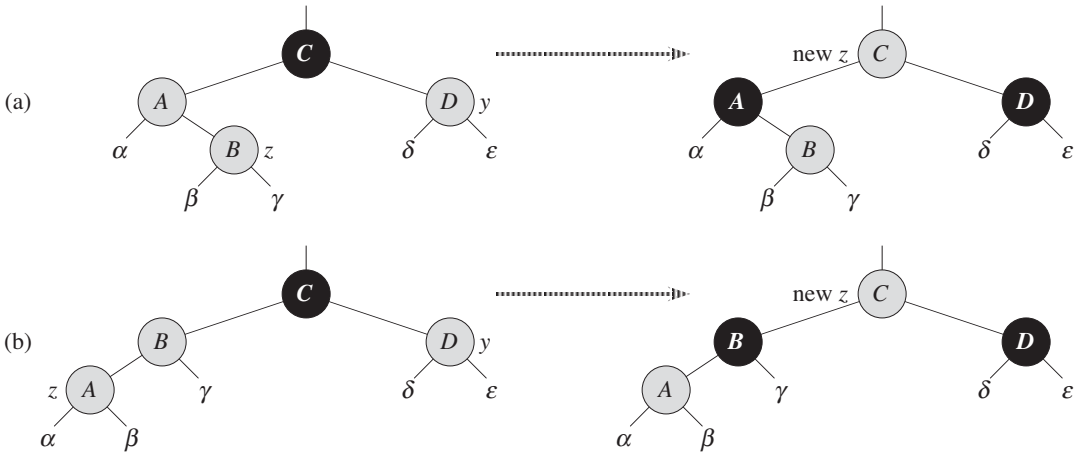


Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z and its parent $z.p$ are both red. We take the same action whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. 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

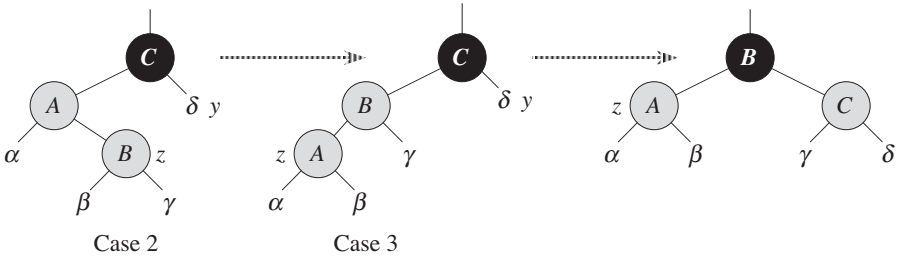


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and 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 ≠ 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 $2 + \text{count}(c)$, 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 $w.left$ 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 $O(\lg n)$.

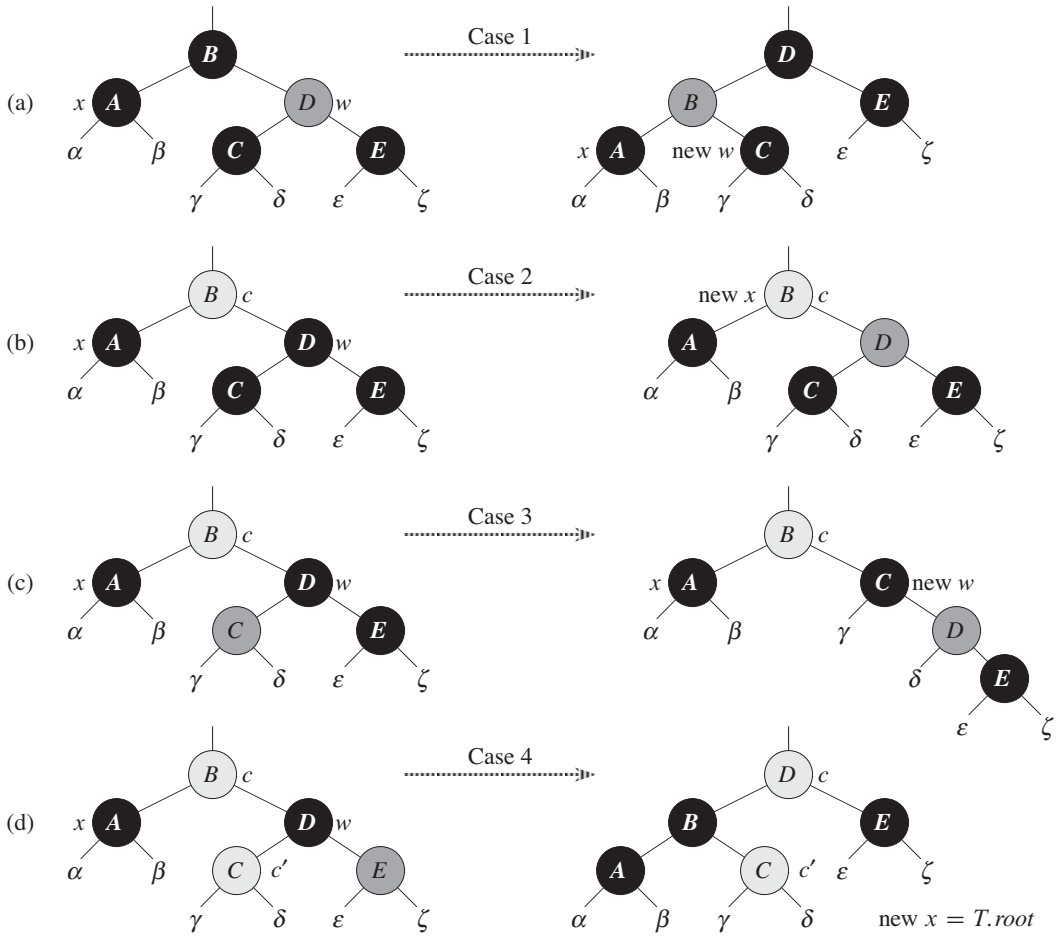


Figure 13.7 The cases in the **while** loop of the procedure RB-DELETE-FIXUP. Darkened nodes have *color* attributes BLACK, heavily shaded nodes have *color* attributes RED, and lightly shaded nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ 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.