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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

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.

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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

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

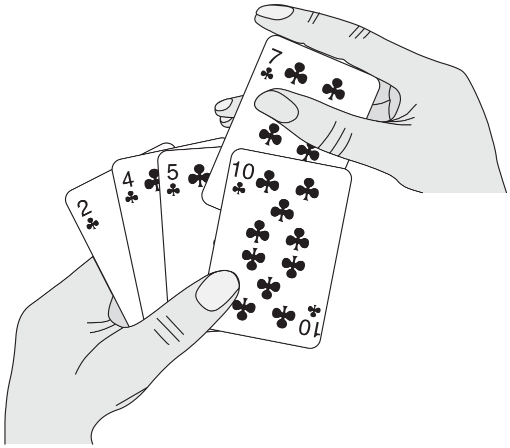


Figure 2.1 Sorting a hand of cards using insertion sort.

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.

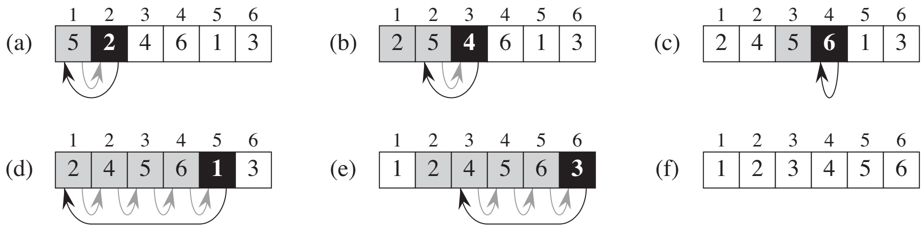


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle 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 $A[1 \mathinner{..} j-1]$ constitutes the currently sorted hand, and the remaining subarray $A[j+1 \mathinner{..} n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 \mathinner{..} j-1]$ are the elements *originally* in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \mathinner{..} j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \mathinner{..} j-1]$ consists of the elements originally in $A[1 \mathinner{..} j-1]$, 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 $A[1 \mathinner{..} j-1]$, 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 $A[1 \mathinner{..} j]$ then consists of the elements originally in $A[1 \mathinner{..} j]$, 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 $j \le A.\mathit{length}$.

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 $j > A.\mathit{length} = n$. 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** $A.\mathit{length}$, and so when this loop terminates, $j = A.\mathit{length} + 1$ (or, equivalently, $j = n + 1$, since $n = A.\mathit{length}$). 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 $A.\mathit{length}$.

  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 ν.

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

Write pseudocode for ***linear search***, which scans through the sequence, looking for ν. 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	$\quad$ $key = A[j]$	c_2	$n-1$
3	$\quad$ // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
4	$\quad$ $i = j-1$	c_4	$n-1$
5	$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$\quad\quad$ $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$\quad\quad$ $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$\quad$ $A[i+1] = key$	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 $A[1 \mathinner{..} j-1]$ to insert element $A[j]$? On average, half the elements in $A[1 \mathinner{..} j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1 \mathinner{..} j-1]$, 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 $A[1 \,..\, j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1 \,..\, j]$.

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 $p \leq q < r$. 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  n1 = q - p + 1
 2  n2 = r - q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 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 $A[p \mathinner{..} q]$, and line 2 computes the length n_2 of the subarray $A[q+1 \mathinner{..} r]$. 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 $A[p \mathinner{..} q]$ into $L[1 \mathinner{..} n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q+1 \mathinner{..} r]$ into $R[1 \mathinner{..} n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illus-

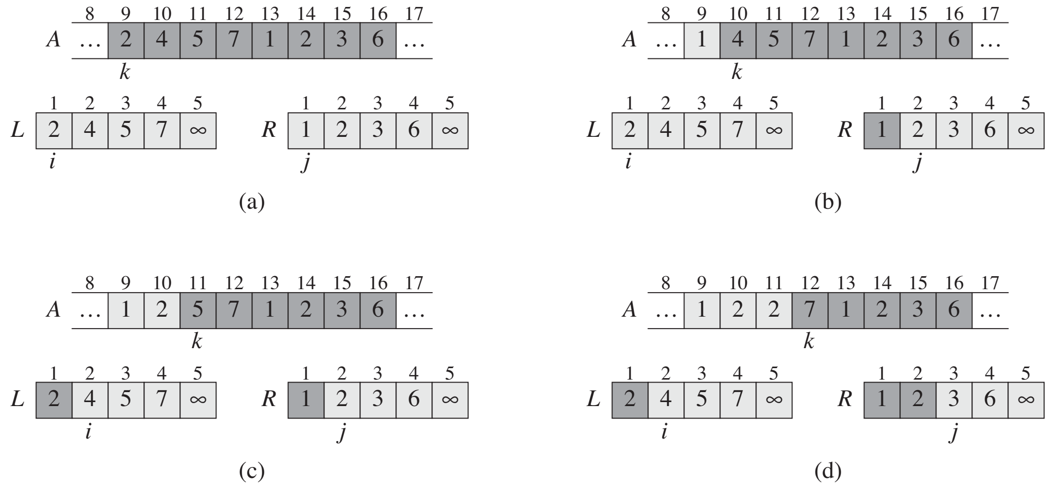


Figure 2.3 The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9\,..\,16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty\rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty\rangle$. 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.

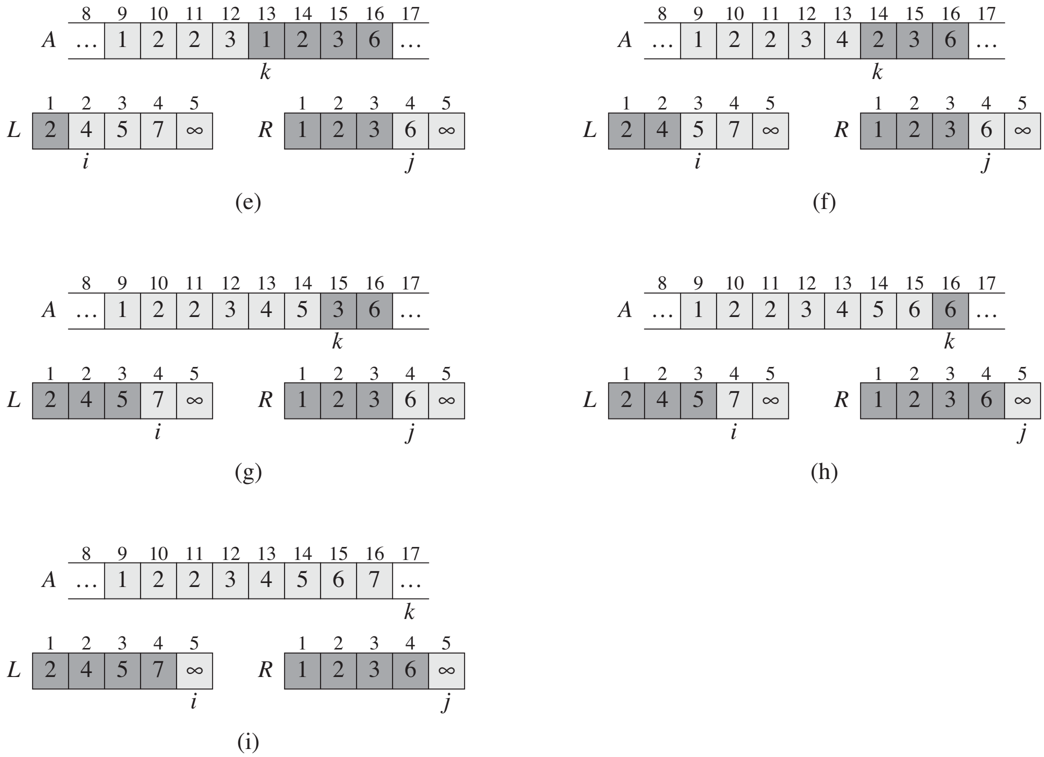


Figure 2.3, continued **(i)** The arrays and indices at termination. At this point, the subarray in $A[9\,..\,16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

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)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    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$(A, 1, A.\mathit{length})$, where once again $A.\mathit{length} = n$. 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.

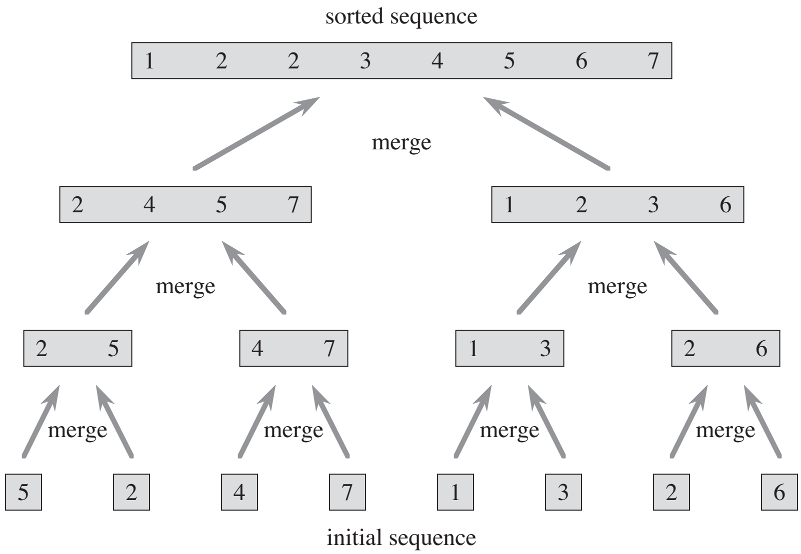


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

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

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

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.

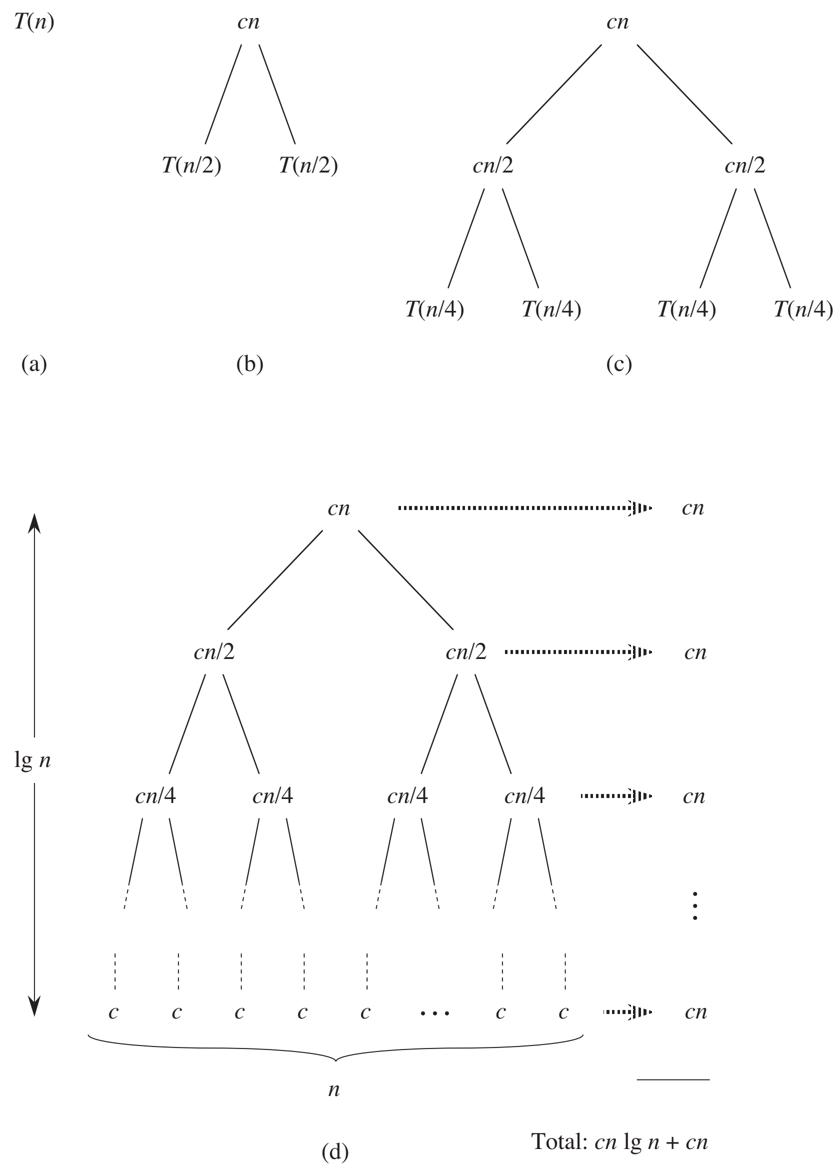


Figure 2.5 How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level 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 ν 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 $A[1 \mathinner{..} n]$ 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 variety 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.

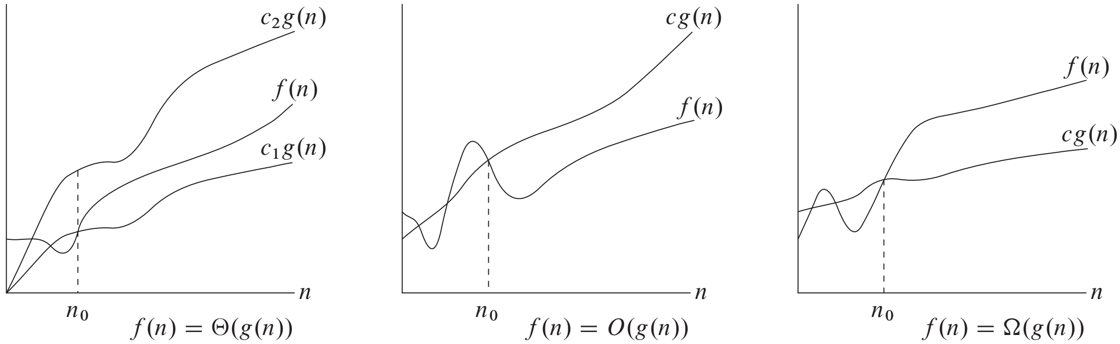
Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand *which* running time we mean. Sometimes we are interested in the worst-case running time. Often, however, we wish to characterize the running time no matter what the input. In other words, we often 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."



$f(n) = \Theta(g(n))$ (a) $f(n) = O(g(n))$ (b) $f(n) = \Omega(g(n))$ (c)

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

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

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

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

We can make the right-hand inequality hold for any value of $n \ge 1$ by choosing any constant $c_2 \ge 1/2$. Likewise, we can make the left-hand inequality hold for any value of $n \ge 7$ by choosing any constant $c_1 \le 1/14$. 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 $6n^3 \ne \Theta(n^2)$. Suppose for the purpose of contradiction that c_2 and n_0 exist such that $6n^3 \le c_2 n^2$ for all $n \ge n_0$. But then dividing by n^2 yields $n \le c_2/6$, 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 $0 \le c_1 n^2 \le an^2 + bn + c \le c_2 n^2$ for all $n \ge n_0$. 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 $2n^2 \ne o(n^2)$.

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

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

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:

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

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:

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

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:

$$\begin{array}{lll} f(n) = O(g(n)) & \text{is like} & a \le b , \\ f(n) = \Omega(g(n)) & \text{is like} & a \ge b , \\ f(n) = \Theta(g(n)) & \text{is like} & a = b , \\ f(n) = o(g(n)) & \text{is like} & a < b , \\ f(n) = \omega(g(n)) & \text{is like} & a > b . \end{array}$$

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

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

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 $f(m) > f(n)$.

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

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

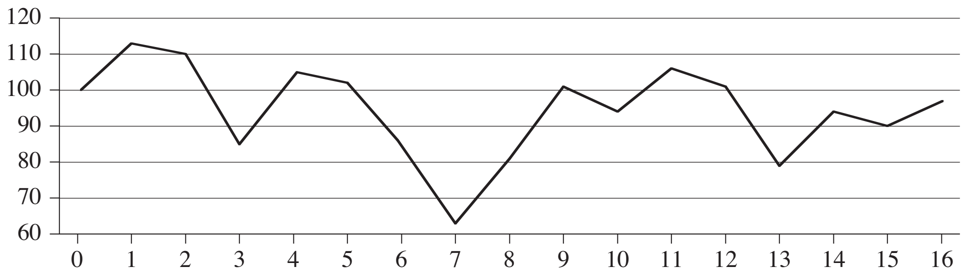
without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

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

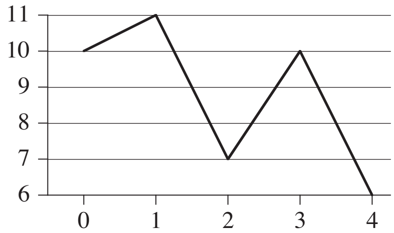
Suppose that you been offered the opportunity to invest in the Volatile Chemical Corporation. Like the chemicals the company produces, the stock price of the Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit of stock only one time and then sell it at a later date, buying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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

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.

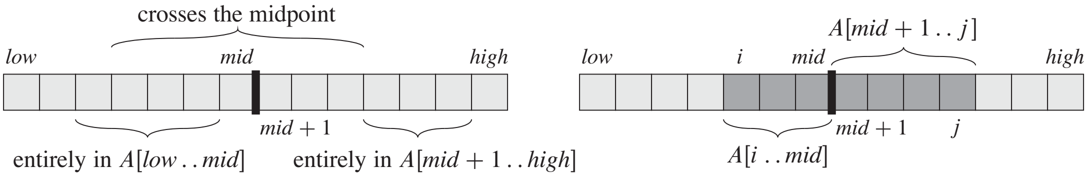
The maximum-subarray problem is interesting only when the array contains some negative numbers. If all the array entries were nonnegative, then the maximum-subarray problem would present no challenge, since the entire array would give the greatest sum.

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



(a) (b)

Figure 4.4 **(a)** Possible locations of subarrays of $A[low \mathinner{..} high]$: entirely in $A[low \mathinner{..} mid]$, entirely in $A[mid+1 \mathinner{..} high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low \mathinner{..} high]$ crossing the midpoint comprises two subarrays $A[i \mathinner{..} mid]$ and $A[mid+1 \mathinner{..} j]$, 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 $A[low \mathinner{..} high]$. 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 $A[i \mathinner{..} mid]$ and $A[mid+1 \mathinner{..} j]$, where $low \le i \le mid$ and $mid < j \le high$. Therefore, we just need to find maximum subarrays of the form $A[i \mathinner{..} mid]$ and $A[mid+1 \mathinner{..} j]$ 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)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
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 $left\text{-}sum + right\text{-}sum$ 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)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

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

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

Analyzing the divide-and-conquer algorithm

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

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

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

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

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

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

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

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

Exercises

4.1-1

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

4.1-2

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

4.1-3

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

4.1-4

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

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

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

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

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

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

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

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

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

Exercises

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

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

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

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

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

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

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

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

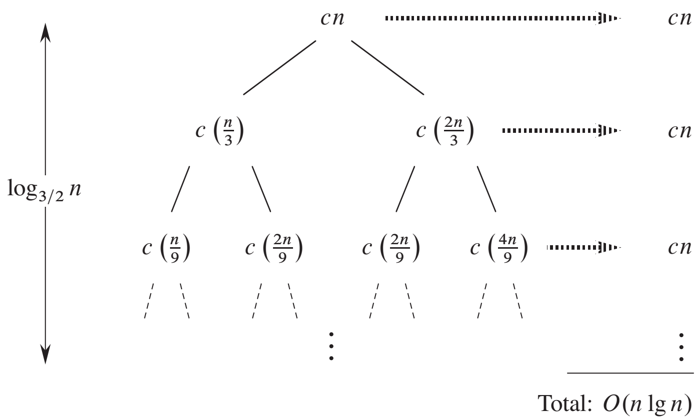


Figure 4.6 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

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

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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

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: $A.\mathit{length}$, which (as usual) gives the number of elements in the array, and $A.\mathit{heap\text{-}size}$, which represents how many elements in the heap are stored within array A. That is, although $A[1 \mathinner{..} A.\mathit{length}]$ may contain numbers, only the elements in $A[1 \mathinner{..} A.\mathit{heap\text{-}size}]$, where $0 \leq A.\mathit{heap\text{-}size} \leq A.\mathit{length}$, 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:

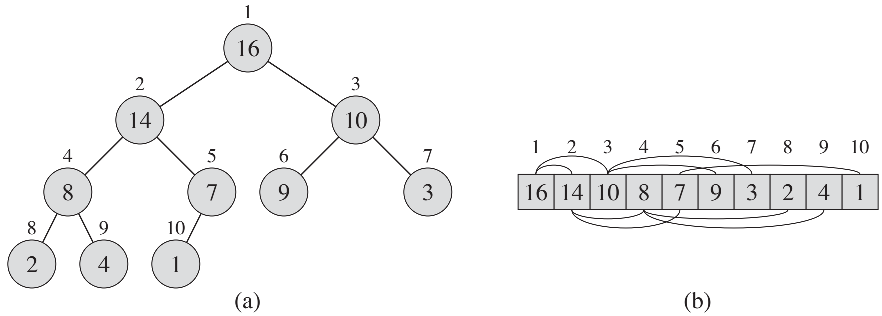


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; 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,

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

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)
l = LEFT(i)
r = RIGHT(i)
if $l \le A.\mathit{heap\text{-}size}$ and $A[l] > A[i]$
    $\mathit{largest} = l$
else $\mathit{largest} = i$
if $r \le A.\mathit{heap\text{-}size}$ and $A[r] > A[\mathit{largest}]$
    $\mathit{largest} = r$
if $\mathit{largest} \ne i$
    exchange $A[i]$ with $A[\mathit{largest}]$
    MAX-HEAPIFY($A, \mathit{largest}$)

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 $A[\mathit{largest}]$, which causes node i and its

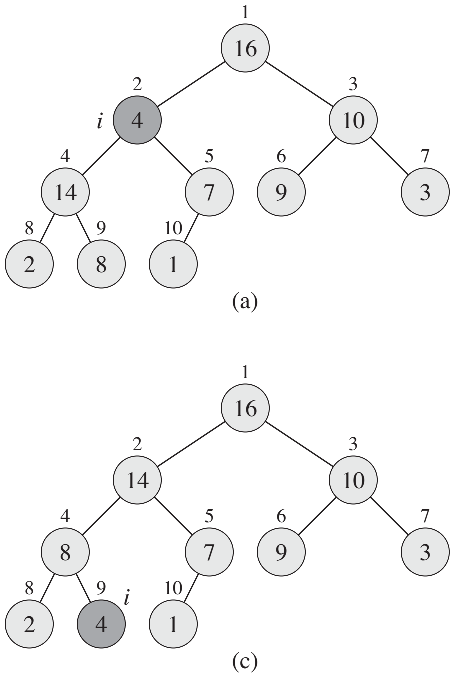


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, 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

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

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

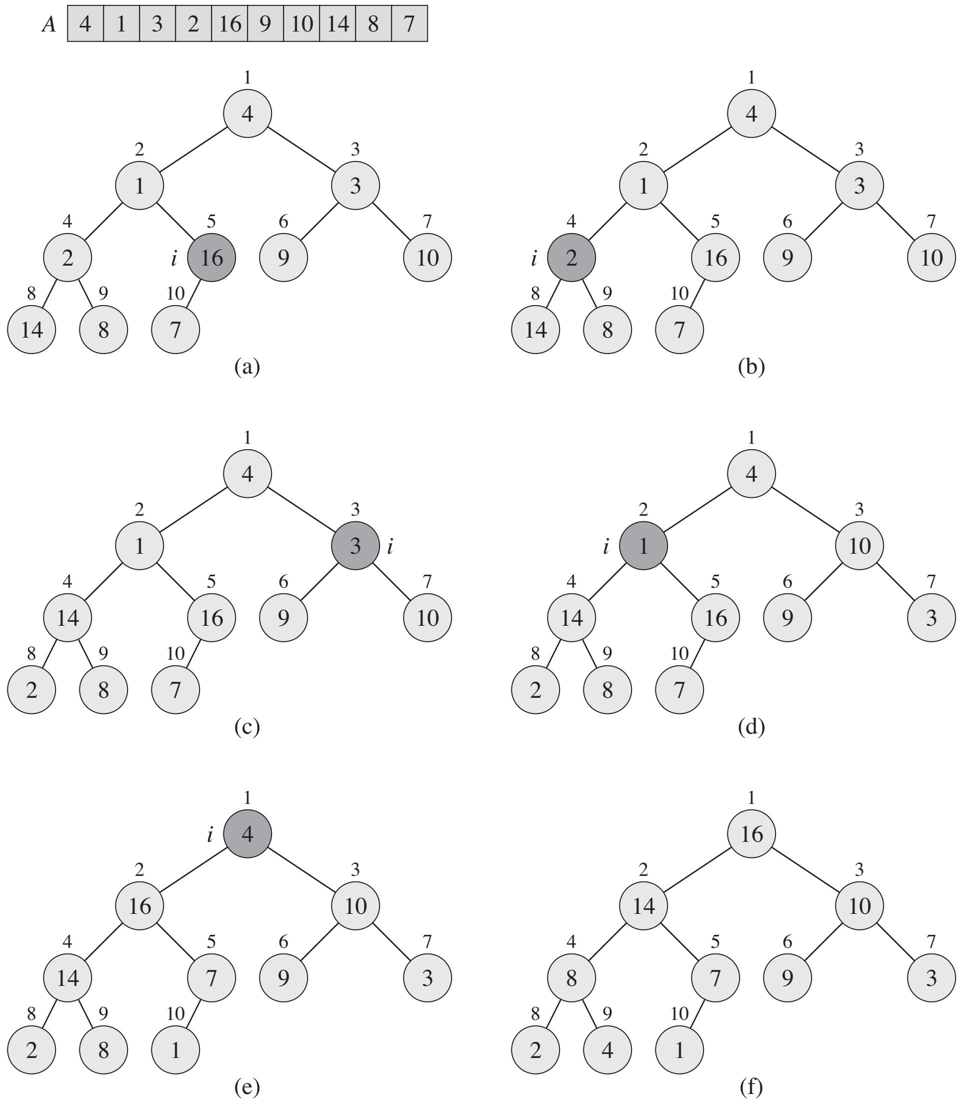


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The 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 $\lfloor A.\mathit{length}/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.\mathit{length}/2 \rfloor$?

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 $n = A.\mathit{length}$. 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 $A[1 \mathinner{..} n-1]$. 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 $A[1 \mathinner{..} i]$ is a max-heap containing the i smallest elements of $A[1 \mathinner{..} n]$, and the subarray $A[i+1 \mathinner{..} n]$ contains the $n-i$ largest elements of $A[1 \mathinner{..} n]$, 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 $\Omega(n \lg n)$.

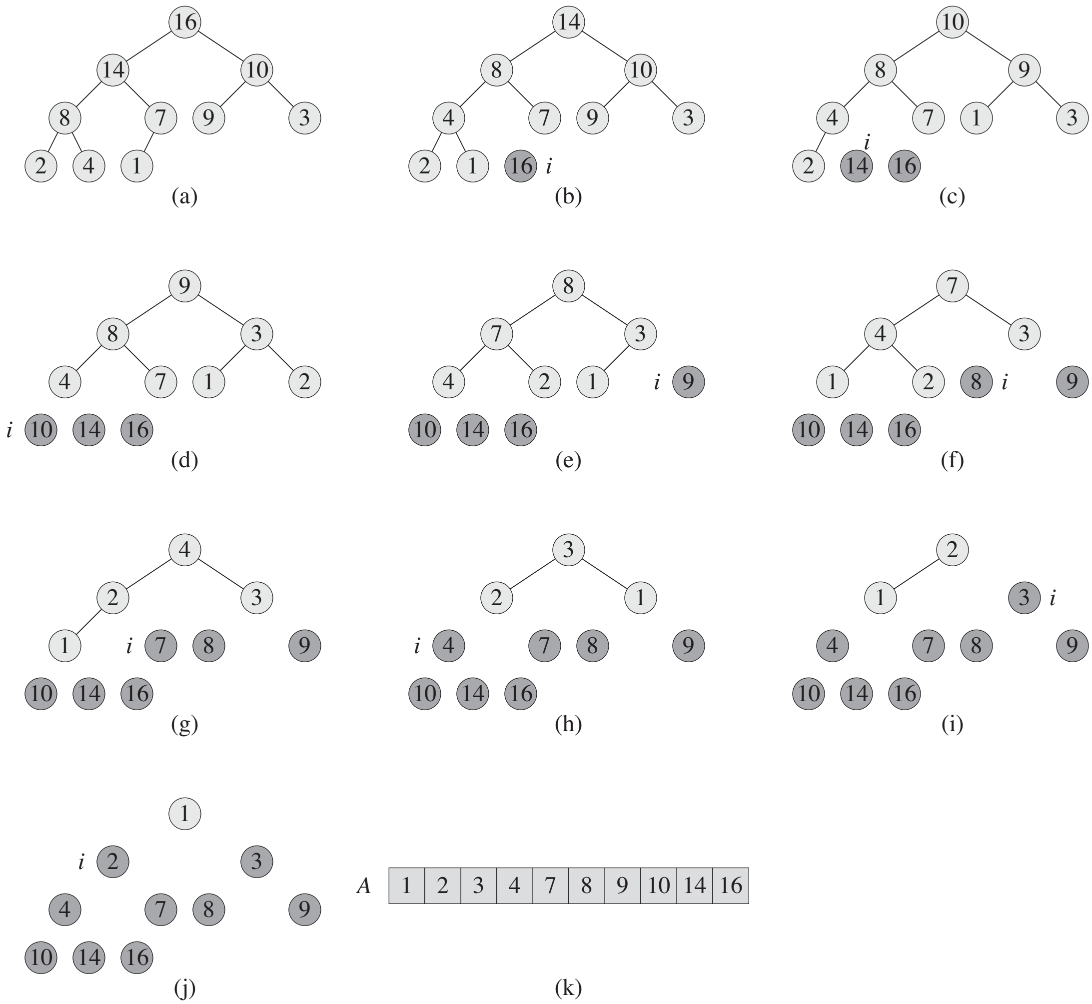


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting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

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)

if $A.\mathit{heap\text{-}size} < 1$
    error "heap underflow"
$\mathit{max} = A[1]$
$A[1] = A[A.\mathit{heap\text{-}size}]$
$A.\mathit{heap\text{-}size} = A.\mathit{heap\text{-}size} - 1$
MAX-HEAPIFY$(A, 1)$
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)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
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 $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

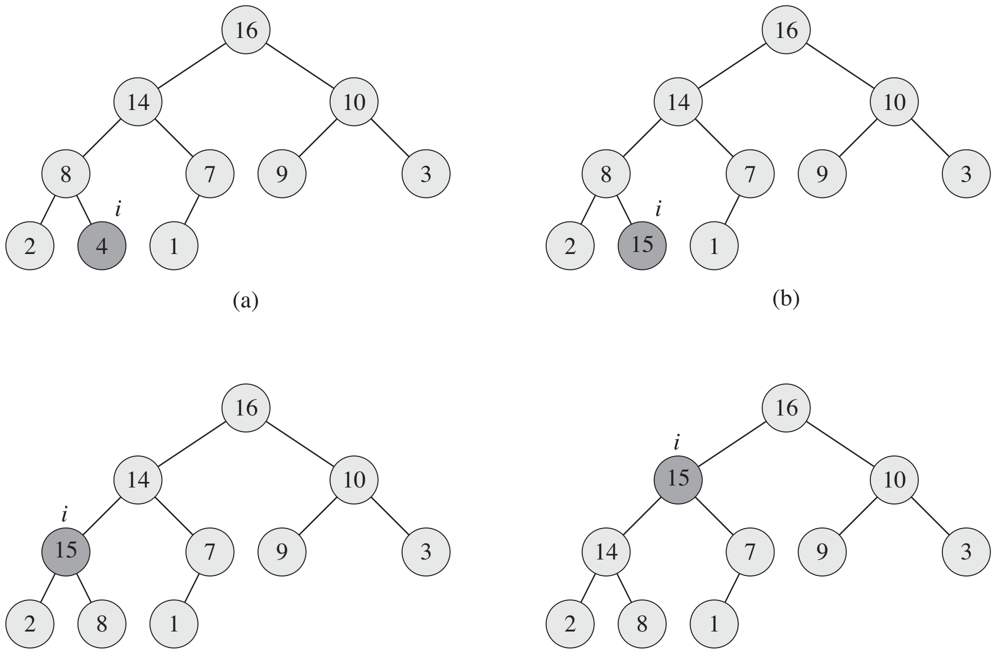


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i 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 $A[1 \mathinner{..} A.\mathit{heap\text{-}size}]$ 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 $A[1 \mathinner{..} A.\mathit{heap\text{-}size}]$ 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)
  A.heap-size = 1
  for i = 2 to A.length
      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 $Y[m,n] < \infty$.

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 $A[p \mathinner{.\,.} r]$:

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

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

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

The following procedure implements quicksort:

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

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

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \mathinner{..} r]$ 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 $A[p \mathinner{..} r]$. 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 $A[k] = x$.

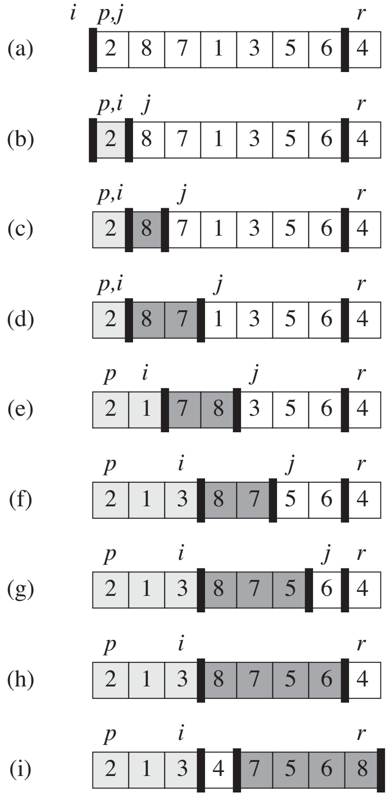


Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have 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.

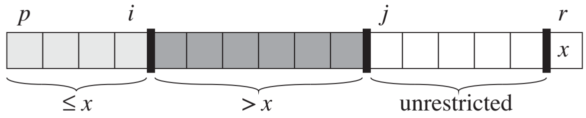


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p\,..\,r]$. The values in $A[p\,..\,i]$ are all less than or equal to x, the values in $A[i+1\,..\,j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j\,..\,r-1]$ can take on any values.

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 $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

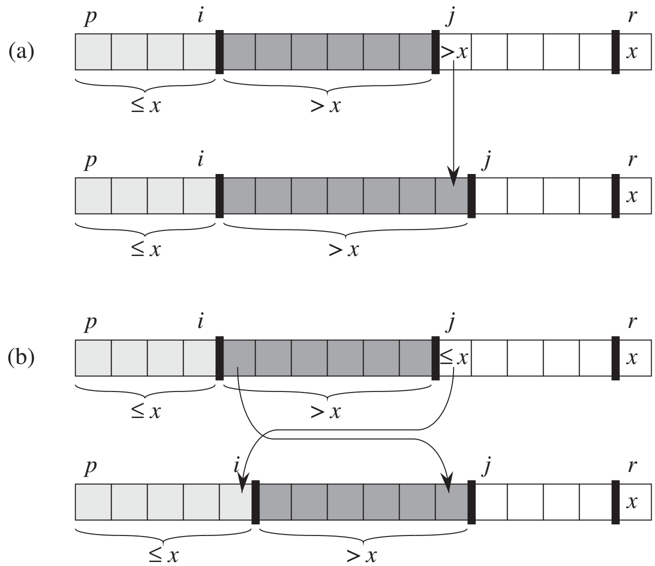


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

7.1-2
What value of q does PARTITION return when all elements in the array $A[p \mathinner{..} r]$ have the same value? Modify PARTITION so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p \mathinner{..} r]$ 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-

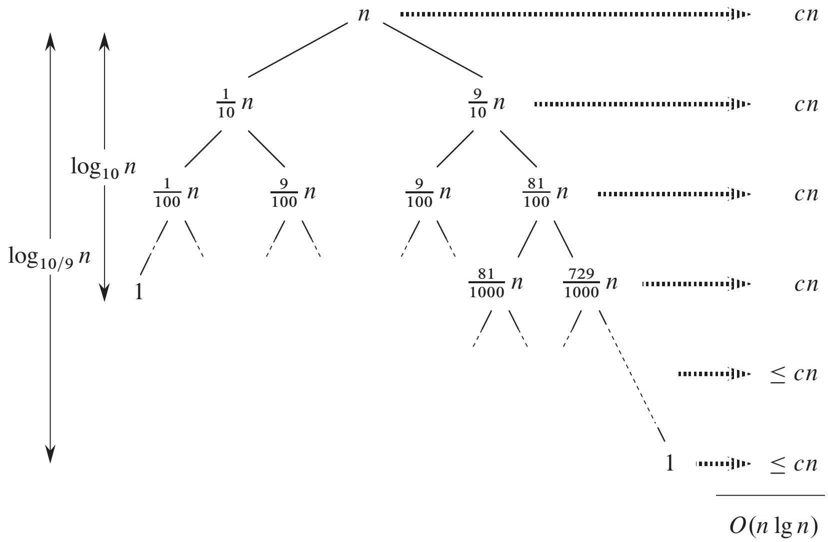


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

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.

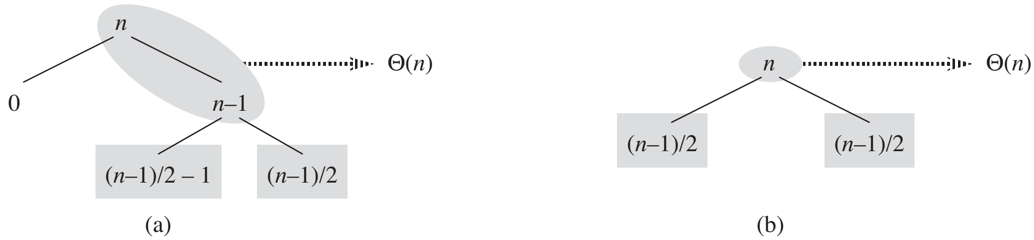


Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n-1$. The partitioning of the subarray of size $n-1$ costs $n-1$ and produces a "good" split: subarrays of size $(n-1)/2-1$ and $(n-1)/2$. **(b)** 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.

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)

$i = $ RANDOM(p, r)
exchange $A[r]$ with $A[i]$
return PARTITION(A, p, r)

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

RANDOMIZED-QUICKSORT(A, p, r)

if $p < r$
    $q = $ RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT$(A, p, q - 1)$
    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,