

---

## 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

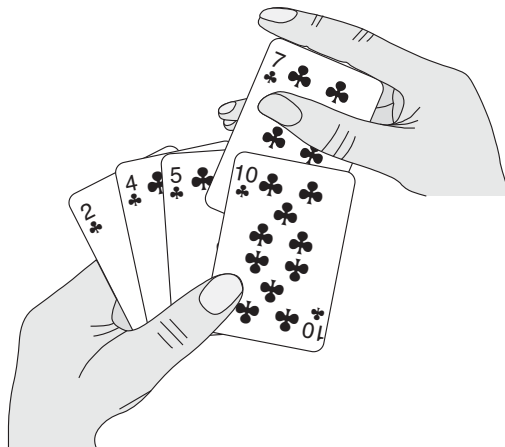
Our first algorithm, insertion sort, solves the *sorting problem* introduced in Chapter 1:

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

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

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

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

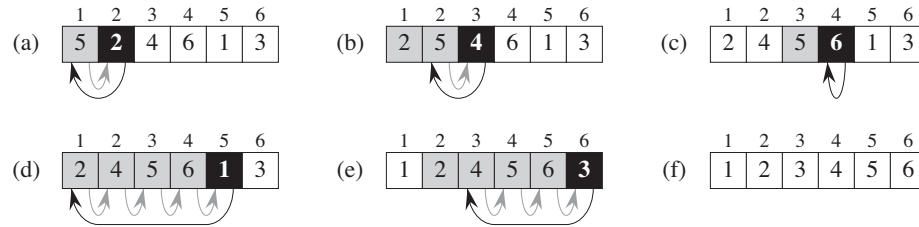


**Figure 2.1** Sorting a hand of cards using insertion sort.

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

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

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



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

#### INSERTION-SORT( $A$ )

```

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

```

#### Loop invariants and the correctness of insertion sort

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

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

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

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

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

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

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

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

Let us see how these properties hold for insertion sort.

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

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

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

---

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

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

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

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

### Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if-else** statements<sup>2</sup> 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.<sup>3</sup>
- 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.<sup>4</sup> In this book, the loop counter retains its value after exiting the loop, unlike some situations that arise in C++, Java, and Pascal. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound. We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for**  $j = 2$  **to**  $A.length$ , and so when this loop terminates,  $j = A.length + 1$  (or, equivalently,  $j = n + 1$ , since  $n = A.length$ ). We use the keyword **to** when a **for** loop increments its loop

---

<sup>2</sup>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.

<sup>3</sup>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.

<sup>4</sup>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  $i$ th 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  $A[1], A[2], \dots, A[j]$ .
- We typically organize compound data into **objects**, which are composed of **attributes**. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, we treat an array as an object with the attribute *length* indicating how many elements it contains. To specify the number of elements in an array  $A$ , we write  $A.length$ .

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

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

Sometimes, a pointer will refer to no object at all. In this case, we give it the special value NIL.

- We pass parameters to a procedure **by value**: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object’s attributes are not. For example, if  $x$  is a parameter of a called procedure, the assignment  $x = y$  within the called procedure is not visible to the calling procedure. The assignment  $x.f = 3$ , however, is visible. Similarly, arrays are passed by pointer, so that

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

- A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single **return** statement.
- The boolean operators “and” and “or” are *short circuiting*. That is, when we evaluate the expression “ $x$  and  $y$ ” we first evaluate  $x$ . If  $x$  evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and so we do not evaluate  $y$ . If, on the other hand,  $x$  evaluates to TRUE, we must evaluate  $y$  to determine the value of the entire expression. Similarly, in the expression “ $x$  or  $y$ ” we evaluate the expression  $y$  only if  $x$  evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as “ $x \neq \text{NIL}$  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 non-decreasing order.

### 2.1-3

Consider the *searching problem*:

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

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

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

### 2.1-4

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

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

---

## 2.2 Analyzing algorithms

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

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

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

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



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

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

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

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

### **Analysis of insertion sort**

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

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

The *running time* of an algorithm on a particular input is the number of primitive operations or “steps” executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the  $i$ th 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.<sup>5</sup>

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  $j = 2, 3, \dots, n$ , 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.

---

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

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes  $c_i$  steps to execute and executes  $n$  times will contribute  $c_i n$  to the total running time.<sup>6</sup> 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

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

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  $j = 2, 3, \dots, n$ , we then find that  $A[i] \leq key$  in line 5 when  $i$  has its initial value of  $j - 1$ . Thus  $t_j = 1$  for  $j = 2, 3, \dots, n$ , 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  $j = 2, 3, \dots, n$ . Noting that

---

<sup>6</sup>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

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

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  $A[1 \dots j - 1]$  to insert element  $A[j]$ ? On average, half the elements in  $A[1 \dots j - 1]$  are less than  $A[j]$ , and half the elements are greater. On average, therefore, we check half of the subarray  $A[1 \dots j - 1]$ , 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  $\Theta$ -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  $\Theta$ -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  $\Theta$ -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  $\Theta$ -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  $A[1 \dots j - 1]$ , we inserted the single element  $A[j]$  into its proper place, yielding the sorted subarray  $A[1 \dots j]$ .

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  $\text{MERGE}(A, p, q, r)$ , 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  $\text{MERGE}$  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  $\infty$  as the sentinel value, so that whenever a card with  $\infty$  is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly  $r - p + 1$  cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

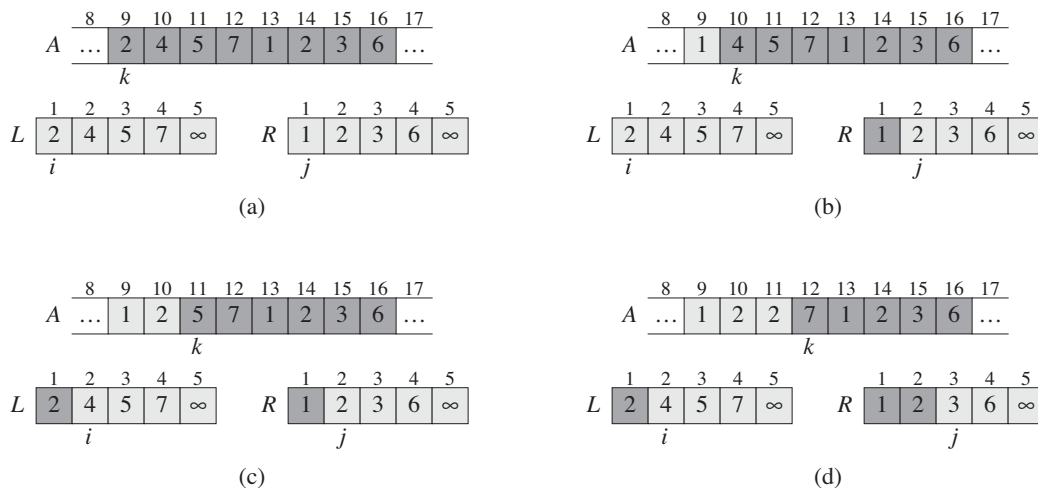
MERGE( $A, p, q, r$ )

```

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

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





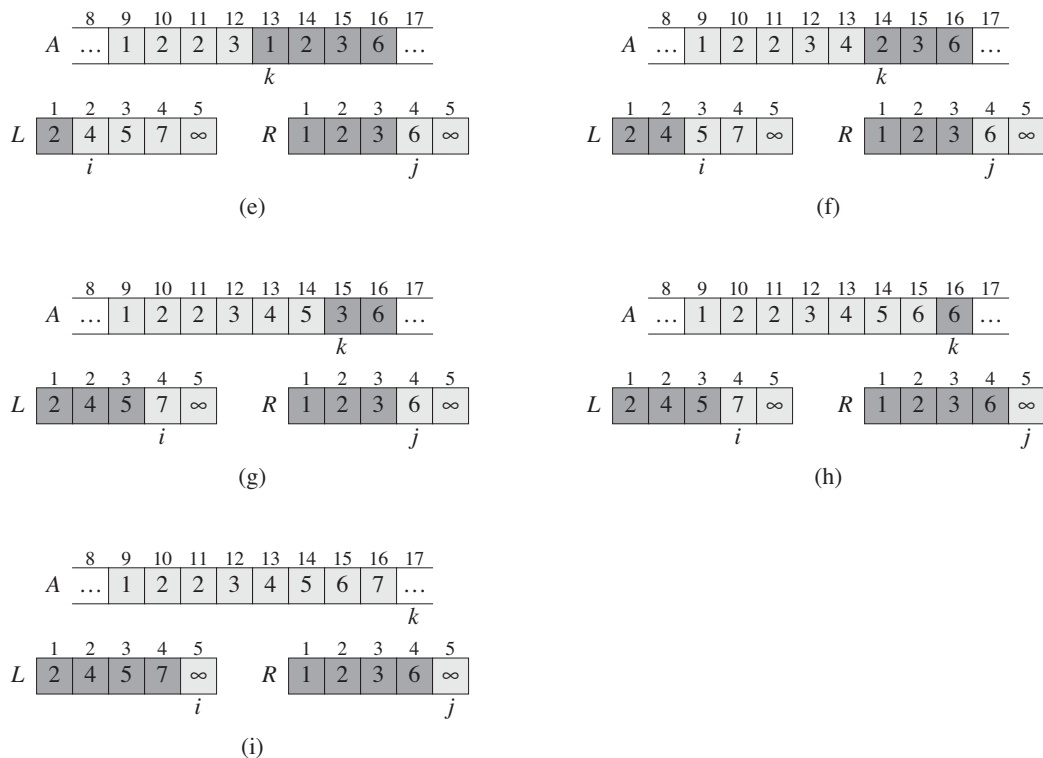
**Figure 2.3** The operation of lines 10–17 in the call `MERGE(A, 9, 12, 16)`, when the subarray  $A[9..16]$  contains the sequence  $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$ . After copying and inserting sentinels, the array  $L$  contains  $\langle 2, 4, 5, 7, \infty \rangle$ , and the array  $R$  contains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

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

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

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

**Initialization:** Prior to the first iteration of the loop, we have  $k = p$ , so that the subarray  $A[p..k - 1]$  is empty. This empty subarray contains the  $k - p = 0$  smallest elements of  $L$  and  $R$ , and since  $i = j = 1$ , both  $L[i]$  and  $R[j]$  are the smallest elements of their arrays that have not been copied back into  $A$ .



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

**Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that  $L[i] \leq R[j]$ . 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,<sup>7</sup> 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.<sup>8</sup>

MERGE-SORT( $A, p, r$ )

```

1  if  $p < r$ 
2       $q = \lfloor (p + r)/2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

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

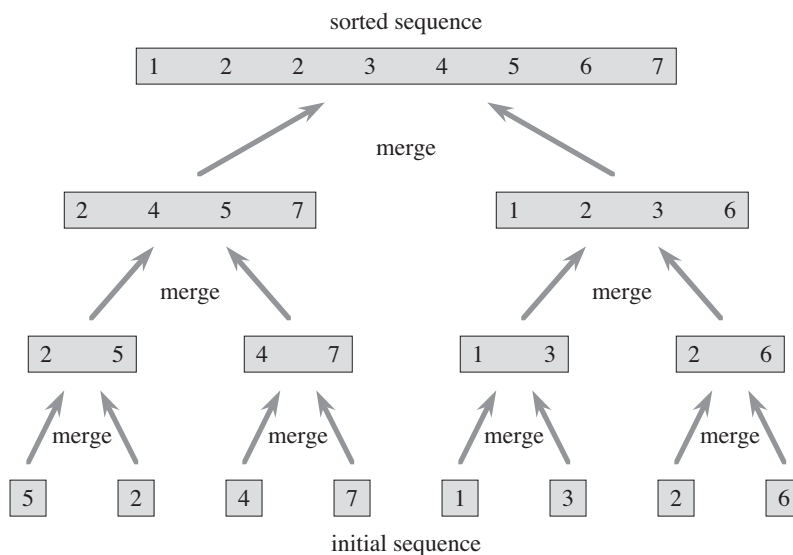
### 2.3.2 Analyzing divide-and-conquer algorithms

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

---

<sup>7</sup>We shall see in Chapter 3 how to formally interpret equations containing  $\Theta$ -notation.

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



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

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

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

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

### Analysis of merge sort

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

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

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

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

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

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

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

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

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

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

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.<sup>9</sup>

---

<sup>9</sup>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  $i$ th 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  $\Theta(n \lg n)$ .

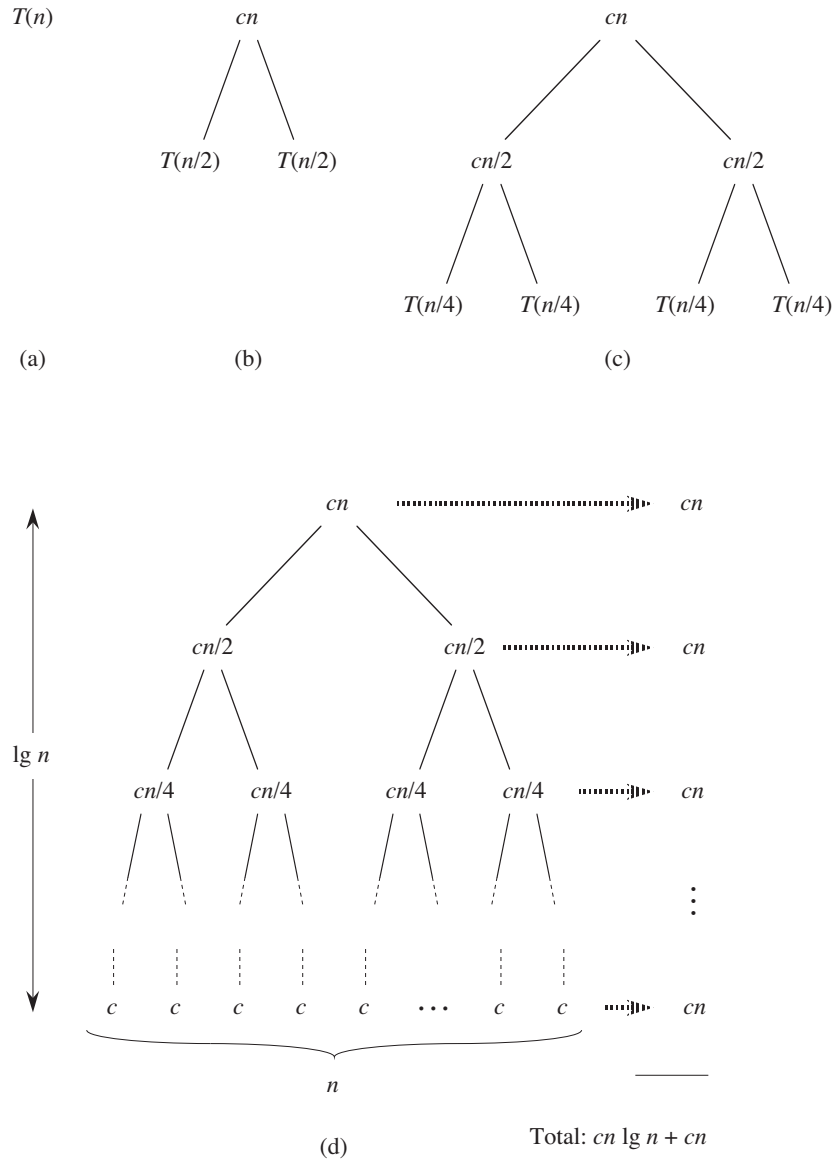
## Exercises

### 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  $A[1 \dots n]$ , we recursively sort  $A[1 \dots n-1]$  and then insert  $A[n]$  into the sorted array  $A[1 \dots n-1]$ . Write a recurrence for the running time of this recursive version of insertion sort.

**2.3-5**

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

**2.3-6**

Observe that the **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray  $A[1 \dots j-1]$ . 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  $\Theta$ -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

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n], \quad (2.3)$$

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  $a_0, a_1, \dots, a_n$  and a value for  $x$ :

```

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

```

- a. In terms of  $\Theta$ -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  $a_0, a_1, \dots, a_n$ .

### 2-4 Inversions

Let  $A[1 \dots n]$  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  $\{1, 2, \dots, n\}$  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  $\Theta$ -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.