2 Getting Started

This chapter will familiarize you with the framework we'll 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 will be introduced in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We'll begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We'll specify algorithms using a pseudocode that should be understandable to you if you have done computer programming. We'll see why insertion sort correctly sorts and analyze its running time. The analysis introduces a notation that describes how running time increases with the number of items to be sorted. Following a discussion of insertion sort, we'll use a method called divide-and-conquer to develop a sorting algorithm called merge sort. We'll end with an analysis of merge sort's running time.

2.1 Insertion sort

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

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

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

The numbers to be sorted are also known as the ***keys***. Although the problem is conceptually about sorting a sequence, the input comes in the form of an array with n elements. When we want to sort numbers, it's often because they are the keys associated with other data, which we call ***satellite data***. Together, a key and satellite data form a ***record***. For example, consider a spreadsheet containing student records with many associated pieces of data such as age, grade-point average, and number of courses taken. Any one of these quantities could be a key, but when the spreadsheet sorts, it moves the

associated record (the satellite data) with the key. When describing a sorting algorithm, we focus on the keys, but it is important to remember that there usually is associated satellite data.

In this book, we'll typically describe algorithms as procedures written in a ***pseudocode*** that is similar in many respects to C, C++, Java, Python,[1] or JavaScript. (Apologies if we've omitted your favorite programming language. We can't list them all.) If you have been introduced to any of these languages, you should have little trouble understanding algorithms "coded" in pseudocode. 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 that looks more like real code. Another difference between pseudocode and real code is that pseudocode often ignores aspects of software engineering—such as data abstraction, modularity, and error handling—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 you might sort a hand of playing cards. Start with an empty left hand and the cards in a pile on the table. Pick up the first card in the pile and hold it with your left hand. Then, with your right hand, remove one card at a time from the pile, and insert it into the correct position in your left hand. As Figure 2.1 illustrates, you find the correct position for a card by comparing it with each of the cards already in your left hand, starting at the right and moving left. As soon as you see a card in your left hand whose value is less than or equal to the card you're holding in your right hand, insert the card that you're holding in your right hand just to the right of this card in your left hand. If all the cards in your left hand have values greater than the card in your right hand, then place this card as the leftmost card in your left hand. At all times, the cards held in your left hand are sorted, and these cards were originally the top cards of the pile on the table.

The pseudocode for insertion sort is given as the procedure INSERTION-SORT on the facing page. It takes two parameters: an array A containing the values to be sorted and the number n of values of sort. The values occupy positions $A[1]$ through $A[n]$ of the array, which we denote by $A[1 : n]$. When the INSERTION-SORT procedure is finished, array $A[1 : n]$ contains the original values, but in sorted order.

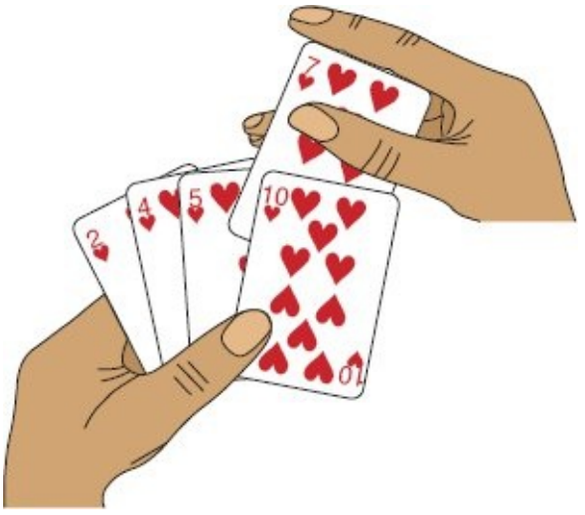

Figure 2.1 Sorting a hand of cards using insertion sort.

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

Loop invariants and the correctness of insertion sort

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

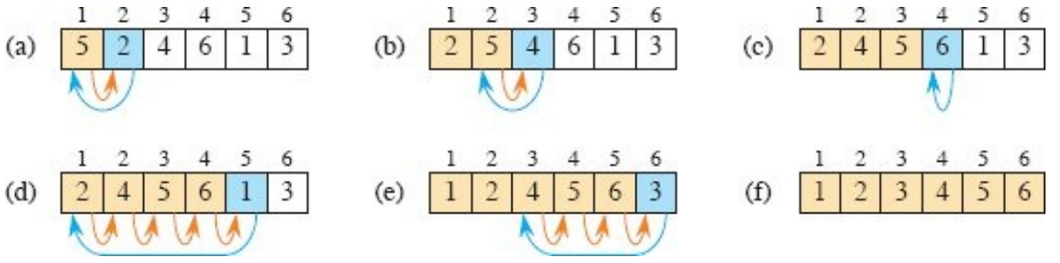

Figure 2.2 The operation of INSERTION-SORT(A, n), where A initially contains the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$ and $n = 6$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of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

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

Loop invariants help us understand why an algorithm is correct. When you're using a loop invariant, you need to show three things:

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: The loop terminates, and when it terminates, the invariant—usually along with the reason that the loop terminated—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, you are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration.) A loop-invariant proof is a form of 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 you are using the loop invariant to show correctness. Typically, you use the loop invariant along with the condition that caused the loop to terminate. Mathematical induction typically applies the inductive step infinitely, but in a loop invariant the "induction" stops when the loop terminates.

Let's see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $i = 2$.[2] The subarray $A[1 : i - 1]$ consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (after

all, how could a subarray with just one value not be sorted?), 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 the values in $A[i-1]$, $A[i-2]$, $A[i-3]$, and so on by one position to the right until it finds the proper position for $A[i]$ (lines 4–7), at which point it inserts the value of $A[i]$ (line 8). The subarray $A[1:i]$ then consists of the elements originally in $A[1:i]$, but in sorted order. ***Incrementing*** i (increasing its value by 1) 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. Let's not get bogged down in such formalism just yet. Instead, we'll rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine loop termination. The loop variable i starts at 2 and increases by 1 in each iteration. Once i's value exceeds n in line 1, the loop terminates. That is, the loop terminates once i equals $n+1$. Substituting $n+1$ for i in the wording of the loop invariant yields that the subarray $A[1:n]$ consists of the elements originally in $A[1:n]$, but in sorted order. Hence, the algorithm is correct.

This method of loop invariants is used to show correctness in various places throughout this book.

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[3] as well. Using indentation instead of textual indicators of block structure, such as **begin** and **end** statements or curly braces, reduces clutter while preserving, or even enhancing, clarity.[4]
- 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 JavaScript.[5] In this book, the loop counter retains its value after the loop is exited, unlike some situations that arise in C++ and Java. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound.[6] We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $i = 2$ **to** n, and so when this loop terminates, i equals $n+1$. We use the keyword **to** when a **for** loop increments its loop counter in each iteration, and we use the keyword **downto** when a **for** loop ***decrements*** its loop

counter (reduces its value by 1 in each iteration). 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.
- Variables (such as i, j, and key) are local to the given procedure. We won't 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.

  Although many programming languages enforce 0-origin indexing for arrays (0 is the smallest valid index), we choose whichever indexing scheme is clearest for human readers to understand. Because people usually start counting at 1, not 0, most—but not all—of the arrays in this book use 1-origin indexing. To be clear about whether a particular algorithm assumes 0-origin or 1-origin indexing, we'll specify the bounds of the arrays explicitly. If you are implementing an algorithm that we specify using 1-origin indexing, but you're writing in a programming language that enforces 0-origin indexing (such as C, C++, Java, Python, or JavaScript), then give yourself credit for being able to adjust. You can either always subtract 1 from each index or allocate each array with one extra position and just ignore position 0.

  The notation ":" denotes a subarray. Thus, $A[i : j]$ indicates the subarray of A consisting of the elements $A[i]$, $A[i + 1]$, … , $A[j]$.[7] We also use this notation to indicate the bounds of an array, as we did earlier when discussing the array $A[1 : n]$.
- 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, if an object x has attribute f, we denote this attribute by $x.f$.

  We treat a variable representing an array or object as a pointer (known as a reference in some programming languages) 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 $y = x$. This way of treating arrays and objects is consistent with most contemporary programming languages.

  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 refers 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 if the calling procedure has a pointer to the same object as x. 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. Again, most contemporary programming languages work this way.
- 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 without having to create objects to package them together.[8]
- The boolean operators "and" and "or" are ***short circuiting***. That is, evaluate the expression "x and y" by first evaluating x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and therefore y is not evaluated. If, on the other hand, x evaluates to TRUE, y must be evaluated to determine the value of the entire expression. Similarly, in the expression "x or y" the expression y is evaluated 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 upon evaluating $x.f$ when x is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called, and the procedure immediately terminates. 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 an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1 : n]$. State a loop invariant for this procedure, and use its

initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in $A[1 : n]$.

```
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3      sum = sum + A[i]
4  return sum
```

2.1-3

Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

2.1-4

Consider the ***searching problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1 : n]$ and a value x.

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

Write pseudocode for ***linear search***, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-5

Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0 : n - 1]$ and $B[0 : n - 1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n + 1)$-element array $C[0 : n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. You might consider resources such as memory, communication bandwidth, or energy consumption. Most often, however, you'll want to measure computational time. If you analyze several candidate algorithms for a problem, you can identify the most efficient one. There might be more than just one viable candidate, but you can often rule out several inferior algorithms in the process.

Before you can analyze an algorithm, you need a model of the technology that it runs

on, including the resources of that technology and a way to express their costs. Most of this book assumes a generic one-processor, ***random-access machine (RAM)*** model of computation as the implementation technology, with the understanding that algorithms are implemented as computer programs. In the RAM model, instructions execute one after another, with no concurrent operations. The RAM model assumes that each instruction takes the same amount of time as any other instruction and that each data access—using the value of a variable or storing into a variable—takes the same amount of time as any other data access. In other words, in the RAM model each instruction or data access takes a constant amount of time—even indexing into an array.[9]

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and 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 you could sort in just one step. Such a RAM would be unrealistic, since such instructions do not appear in real computers. 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).

The data types in the RAM model are integer, floating point (for storing real-number approximations), and character. Real computers do not usually have a separate data type for the boolean values TRUE and FALSE. Instead, they often test whether an integer value is 0 (FALSE) or nonzero (TRUE), as in C. Although we typically do not concern ourselves with precision for floating-point values in this book (many numbers cannot be represented exactly in floating point), precision is crucial for most applications. We also assume that each word of data has a limit on the number of bits. For example, when working with inputs of size n, we typically assume that integers are represented by $c \log_2 n$ 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—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: to compute x^n when x and n are general integers typically takes time logarithmic in n (see equation (31.34) on page 934), and you must worry about whether the result fits into a computer word. If n is an exact power of 2, however, exponentiation can usually be viewed as a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by n positions to the left. In most computers, shifting the bits of an integer by 1 position to the left is equivalent to multiplying by 2, so that shifting the bits by n positions to the left is equivalent to multiplying by 2^n. Therefore, such computers can

compute 2^n in 1 constant-time instruction by shifting the integer 1 by n positions to the left, as long as n is no more than the number of bits in a computer word. We'll try to avoid such gray areas in the RAM model and treat computing 2^n and multiplying by 2^n as constant-time operations when the result is small enough to fit in a computer word.

The RAM model does not account for the memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. Several other computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. Section 11.5 and a handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book do 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.

Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because an algorithm might behave differently for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Analysis of insertion sort

How long does the INSERTION-SORT procedure take? One way to tell would be for you to run it on your computer and time how long it takes to run. Of course, you'd first have to implement it in a real programming language, since you cannot run our pseudocode directly. What would such a timing test tell you? You would find out how long insertion sort takes to run on your particular computer, on that particular input, under the particular implementation that you created, with the particular compiler or interpreter that you ran, with the particular libraries that you linked in, and with the particular background tasks that were running on your computer concurrently with your timing test (such as checking for incoming information over a network). If you run insertion sort again on your computer with the same input, you might even get a different timing result. From running just one implementation of insertion sort on just one computer and on just one input, what would you be able to determine about insertion sort's running time if you were to give it a different input, if you were to run it on a different computer, or if you were to implement it in a different programming language? Not much. We need a way to predict, given a new input, how long insertion sort will take.

Instead of timing a run, or even several runs, of insertion sort, we can determine how long it takes by analyzing the algorithm itself. We'll examine how many times it executes each line of pseudocode and how long each line of pseudocode takes to run. We'll first come up with a precise but complicated formula for the running time. Then, we'll distill the important part of the formula using a convenient notation that can help us compare

the running times of different algorithms for the same problem.

How do we analyze insertion sort? First, let's acknowledge that the running time depends on the input. You shouldn't be terribly surprised that sorting a thousand numbers takes longer than sorting three numbers. Moreover, insertion sort can take different amounts of time to sort two input arrays of the same size, depending on how nearly sorted they already are. Even though the running time can depend on many features of the input, we'll focus on the one that has been shown to have the greatest effect, namely the size of the input, and 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 "input size" more carefully. We also need to be clear about whether we are discussing the running time for an input that elicits the worst-case behavior, the best-case behavior, or some other case.

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 number n of items being sorted. 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 more than just one number. For example, if the input to an algorithm is a graph, we usually characterize the input size by both the number of vertices and the number of edges in the graph. We'll 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 instructions and data accesses executed. How we account for these costs should be independent of any particular computer, but within the framework of the RAM model. 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 might take more or less time than another line, but we'll assume that each execution of the kth line takes c_k time, where c_k 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.[10]

Let's analyze the INSERTION-SORT procedure. As promised, we'll start by devising a precise formula that uses the input size and all the statement costs c_k. This formula turns out to be messy, however. We'll then switch to a simpler notation that is more concise and easier to use. This simpler notation makes clear how to compare the running times of algorithms, especially as the size of the input increases.

To analyze the INSERTION-SORT procedure, let's view it on the following page with the time cost of each statement and the number of times each statement is executed. For each $i = 2, 3, \ldots, n$, let t_i denote the number of times the **while** loop test in line 5 is executed for that value of i. When a **for** or **while** loop exits in the usual way—because the test in the loop header comes up FALSE—the test is executed one time more than the loop body. Because comments are not executable statements, assume that they take no

time.

The running time of the algorithm is the sum of running times for each statement executed. A statement that takes c_k steps to execute and executes m times contributes $c_k m$ to the total running time.[11] We usually denote the running time of an algorithm on an input of size n by $T(n)$. 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

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

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) .$$

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 when the array is already sorted. In this case, each time that line 5 executes, the value of *key*—the value originally in $A[i]$—is already greater than or equal to all values in $A[1 : i-1]$, so that the **while** loop of lines 5–7 always exits upon the first test in line 5. Therefore, we have that $t_i = 1$ for $i = 2, 3, \ldots, n$, and the best-case running time is given by

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned} \tag{2.1}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_k (where $a = c_1 + c_2 + c_4 + c_5 + c_8$ and $b = c_2 + c_4 + c_5 + c_8$). The running time is thus a ***linear function*** of n.

The worst case arises when the array is in reverse sorted order—that is, it starts out in decreasing order. The procedure must compare each element $A[i]$ with each element in the entire sorted subarray $A[1 : i-1]$, and so $t_i = i$ for $i = 2, 3, \ldots, n$. (The procedure finds that $A[j] > key$ every time in line 5, and the **while** loop exits only when j reaches 0.) Noting that

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1$$
$$= \frac{n(n+1)}{2} - 1 \quad \text{(by equation (A.2) on page 1141)}$$

and

$$\sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i$$
$$= \frac{n(n-1)}{2} \quad \text{(again, by equation (A.2))},$$

we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned} \tag{2.2}$$

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_k (now, $a = c_5/2 + c_6/2 + c_7/2$, $b = c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8$, and $c = -(c_2 + c_4 + c_5 + c_8)$). The running time 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 we'll also see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

Our analysis of insertion sort 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'll usually (but not always) concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size n. Why? Here are three reasons:

- The worst-case running time of an algorithm gives an upper bound on the running time for *any* input. If you know it, then you have a guarantee that the algorithm never takes any longer. You need not make some educated guess about the running time and hope that it never gets much worse. This feature is especially important for real-time computing, in which operations must complete by a

deadline.

- 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 often occurs 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 you run insertion sort on an array of n randomly chosen numbers. How long does it take to determine where in subarray $A[1 : i - 1]$ to insert element $A[i]$? On average, half the elements in $A[1 : i - 1]$ are less than $A[i]$, and half the elements are greater. On average, therefore, $A[i]$ is compared with just half of the subarray $A[1 : i - 1]$, and so t_i is about $i/2$. 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'll be interested in the ***average-case*** running time of an algorithm. We'll 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'll 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

In order to ease our analysis of the INSERTION-SORT procedure, we used some simplifying abstractions. First, we ignored the actual cost of each statement, using the constants c_k to represent these costs. Still, the best-case and worst-case running times in equations (2.1) and (2.2) are rather unwieldy. The constants in these expressions give us more detail than we really need. That's why we also expressed the best-case running time as $an + b$ for constants a and b that depend on the statement costs c_k and why we expressed the worst-case running time as $an^2 + bn + c$ for constants a, b, and c that depend on the statement costs. We thus ignored not only the actual statement costs, but also the abstract costs c_k.

Let's 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's worst-case running time, when we ignore the lower-order terms and the leading term's constant coefficient, only the factor of n^2 from the leading term remains. That factor, n^2, is by far the most

important part of the running time. For example, suppose that an algorithm implemented on a particular machine takes $n^2/100 + 100n + 17$ microseconds on an input of size n. Although the coefficients of 1/100 for the n^2 term and 100 for the n term differ by four orders of magnitude, the $n^2/100$ term dominates the $100n$ term once n exceeds 10,000. Although 10,000 might seem large, it is smaller than the population of an average town. Many real-world problems have much larger input sizes.

To highlight the order of growth of the running time, we have a special notation that uses the Greek letter Θ (theta). We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared" or just "theta n-squared"). We also write that insertion sort has a best-case running time of $\Theta(n)$ ("theta of n" or "theta n"). For now, think of Θ-notation as saying "roughly proportional when n is large," so that $\Theta(n^2)$ means "roughly proportional to n^2 when n is large" and $\Theta(n)$ means "roughly proportional to n when n is large" We'll use Θ-notation informally in this chapter and 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 on large enough inputs, an algorithm whose worst-case running time is $\Theta(n^2)$, for example, takes less time in the worst case than an algorithm whose worst-case running time is $\Theta(n^3)$. Regardless of the constants hidden by the Θ-notation, there is always some number, say n_0, such that for all input sizes $n \geq n_0$, the $\Theta(n^2)$ algorithm beats the $\Theta(n^3)$ algorithm in the worst case.

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 $A[1 : n]$ by first finding the smallest element of $A[1 : n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2 : n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3 : n]$, and exchange it with $A[3]$. 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 worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

2.2-3

Consider linear search again (see Exercise 2.1-4). How many elements of the input array

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? Using Θ-notation, give the average-case and worst-case running times of linear search. Justify your answers.

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

2.3 Designing algorithms

You can choose from a wide range of algorithm design techniques. Insertion sort uses the ***incremental*** method: for each element $A[i]$, insert it into its proper place in the subarray $A[1 : i]$, having already sorted the subarray $A[1 : i - 1]$.

This section examines another design method, known as "divide-and-conquer," which we 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 using an algorithm that follows the divide-and-conquer method is that analyzing its running time is often straightforward, using techniques that we'll explore in Chapter 4.

2.3.1 The divide-and-conquer method

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they ***recurse*** (call themselves) one or more times to handle closely related subproblems. These algorithms typically follow the ***divide-and-conquer*** method: 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.

In the divide-and-conquer method, if the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer method. In each step, it sorts a subarray $A[p : r]$, starting with the entire array $A[1 : n]$ and recursing down to smaller and smaller subarrays. Here is how merge sort operates:

Divide the subarray $A[p : r]$ to be sorted into two adjacent subarrays, each of half the size. To do so, compute the midpoint q of $A[p : r]$ (taking the average of p and r), and

divide $A[p : r]$ into subarrays $A[p : q]$ and $A[q + 1 : r]$.

Conquer by sorting each of the two subarrays $A[p : q]$ and $A[q + 1 : r]$ recursively using merge sort.

Combine by merging the two sorted subarrays $A[p : q]$ and $A[q + 1 : r]$ back into $A[p : r]$, producing the sorted answer.

The recursion "bottoms out"—it reaches the base case—when the subarray $A[p : r]$ to be sorted has just 1 element, that is, when p equals r. As we noted in the initialization argument for INSERTION-SORT's loop invariant, a subarray comprising just a single element is always sorted.

The key operation of the merge sort algorithm occurs in the "combine" step, which merges two adjacent, sorted subarrays. The merge operation is performed by the auxiliary procedure MERGE(A, p, q, r) on the following page, where A is an array and p, q, and r are indices into the array such that $p \le q < r$. The procedure assumes that the adjacent subarrays $A[p : q]$ and $A[q + 1 : r]$ were already recursively sorted. It ***merges*** the two sorted subarrays to form a single sorted subarray that replaces the current subarray $A[p : r]$.

To understand how the MERGE procedure works, let's return to our card-playing motif. Suppose that you have two piles of cards face up on a table. Each pile is sorted, with the smallest-value cards on top. You wish to merge the two piles into a single sorted output pile, which is to be face down on the table. The 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. Repeat this step until one input pile is empty, at which time you can just take the remaining input pile and flip over the entire pile, placing it face down onto the output pile.

Let's think about how long it takes to merge two sorted piles of cards. Each basic step takes constant time, since you are comparing just the two top cards. If the two sorted piles that you start with each have $n/2$ cards, then the number of basic steps is at least $n/2$ (since in whichever pile was emptied, every card was found to be smaller than some card from the other pile) and at most n (actually, at most $n - 1$, since after $n - 1$ basic steps, one of the piles must be empty). With each basic step taking constant time and the total number of basic steps being between $n/2$ and n, we can say that merging takes time roughly proportional to n. That is, merging takes $\Theta(n)$ time.

In detail, the MERGE procedure works as follows. It copies the two subarrays $A[p : q]$ and $A[q + 1 : r]$ into temporary arrays L and R ("left" and "right"), and then it merges the values in L and R back into $A[p : r]$. Lines 1 and 2 compute the lengths n_L and n_R of the subarrays $A[p : q]$ and $A[q + 1 : r]$, respectively. Then line 3 creates arrays $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$ with respective lengths n_L and n_R.[12] The **for** loop of lines 4–5 copies the subarray $A[p : q]$ into L, and the **for** loop of lines 6–7 copies the subarray $A[q + 1 : r]$

into R.

```
MERGE(A, p, q, r)
 1  n_L = q – p + 1              // length of A[p : q]
 2  n_R = r – q                  // length of A[q + 1 : r]
 3  let L[0 : n_L – 1] and R[0 : n_R – 1] be new arrays
 4  for i = 0 to n_L – 1         // copy A[p : q] into L[0 : n_L – 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R – 1         // copy A[q + 1 : r] into R[0 : n_R – 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                        // i indexes the smallest remaining element in L
 9  j = 0                        // j indexes the smallest remaining element in R
10  k = p                        // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_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
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Lines 8–18, illustrated in Figure 2.3, perform the basic steps. The **while** loop of lines 12–18 repeatedly identifies the smallest value in L and R that has yet to be copied back into $A[p : r]$ and copies it back in. As the comments indicate, the index k gives the position of A that is being filled in, and the indices i and j give the positions in L and R, respectively, of the smallest remaining values. Eventually, either all of L or all of R is copied back into $A[p : r]$, and this loop terminates. If the loop terminates because all of R has been copied back, that is, because j equals n_R, then i is still less than n_L, so that

some of L has yet to be copied back, and these values are the greatest in both L and R. In this case, the **while** loop of lines 20–23 copies these remaining values of L into the last few positions of $A[p : r]$. Because j equals n_R, the **while** loop of lines 24–27 iterates 0 times. If instead the **while** loop of lines 12–18 terminates because i equals n_L, then all of L has already been copied back into $A[p : r]$, and the **while** loop of lines 24–27 copies the remaining values of R back into the end of $A[p : r]$.

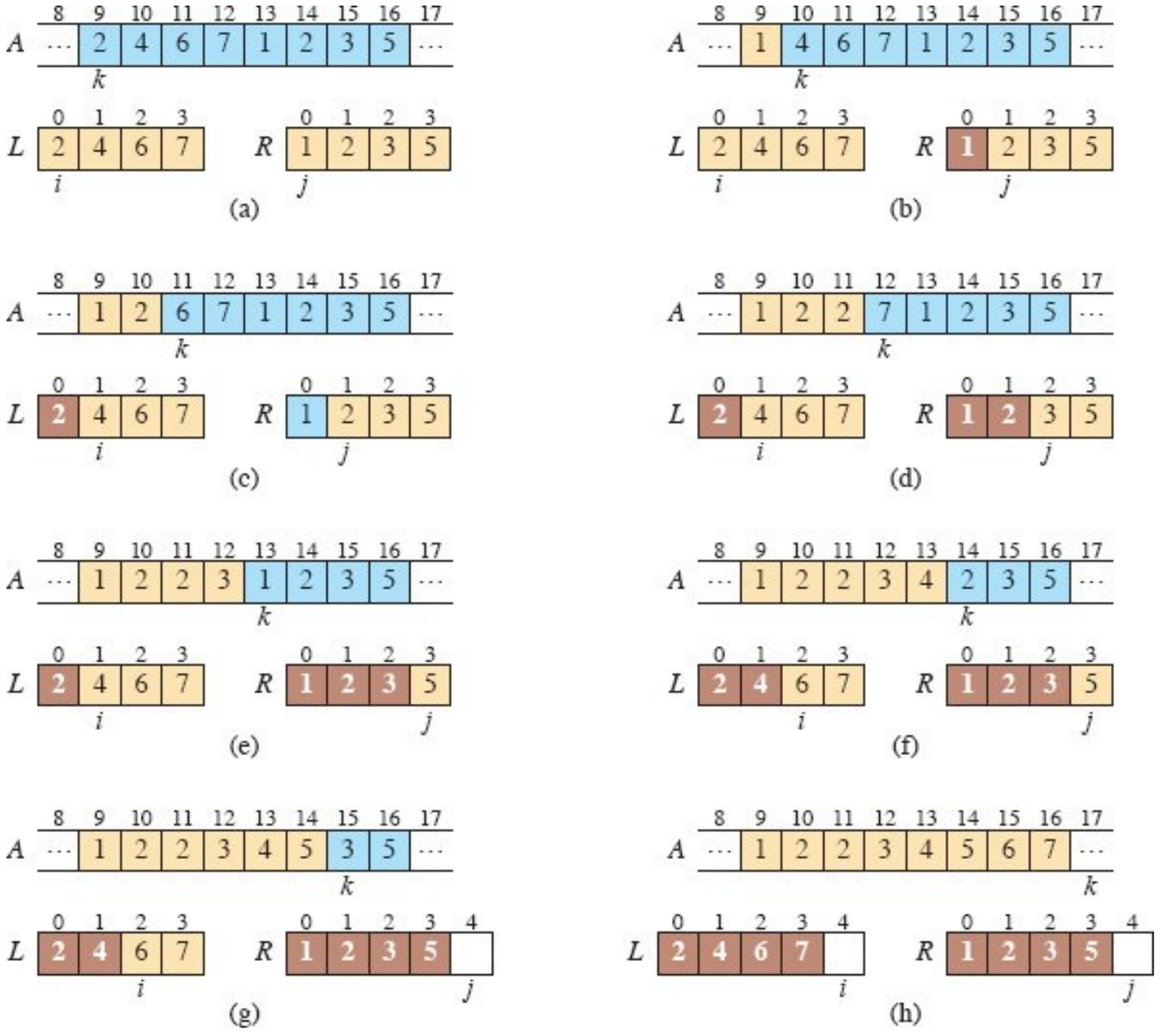

Figure 2.3 The operation of the **while** loop in lines 8–18 in the call MERGE(A, 9, 12, 16), when the subarray $A[9 : 16]$ contains the values $\langle 2, 4, 6, 7, 1, 2, 3, 5\rangle$. After allocating and copying into the arrays L and R, the array L contains $\langle 2, 4, 6, 7\rangle$, and the array R contains $\langle 1, 2, 3, 5\rangle$. Tan positions in A contain their final values, and tan positions in L and R contain values that have yet to be copied back into A. Taken together, the tan positions always comprise the values originally in $A[9 : 16]$. Blue positions in A contain values that will be copied over, and dark positions in L and R contain values that have already been copied back into A. **(a)–(g)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–18. At the point in part (g), all values in R have been copied back into A (indicated by j equaling the length of R), and so the **while** loop in lines 12–18 terminates. **(h)** The arrays and indices at termination. The **while** loops of lines 20–23 and 24–27 copied back into A the remaining values in L and R, which are the largest values originally in $A[9 : 16]$. Here, lines 20–23 copied $L[2 : 3]$ into $A[15 : 16]$, and because all values in R had already been copied back into A, the **while** loop of lines 24–27 iterated 0 times. At this point, the subarray in $A[9 : 16]$ is sorted.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$,[13] observe

that each of lines 1–3 and 8–10 takes constant time, and the **for** loops of lines 4–7 take $\Theta(n_L + n_R) = \Theta(n)$ time.[14] To account for the three **while** loops of lines 12–18, 20–23, and 24–27, observe that each iteration of these loops copies exactly one value from L or R back into A and that every value is copied back into A exactly once. Therefore, these three loops together make a total of n iterations. Since each iteration of each of the three loops takes constant time, the total time spent in these three loops is $\Theta(n)$.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) on the facing page sorts the elements in the subarray $A[p : r]$. If p equals r, the subarray has just 1 element and is therefore already sorted. Otherwise, we must have $p < r$, and MERGE-SORT runs the divide, conquer, and combine steps. The divide step simply computes an index q that partitions $A[p : r]$ into two adjacent subarrays: $A[p : q]$, containing $\lceil n/2 \rceil$ elements, and $A[q + 1 : r]$, containing $\lfloor n/2 \rfloor$ elements.[15] The initial call MERGE-SORT$(A, 1, n)$ sorts the entire array $A[1 : n]$.

Figure 2.4 illustrates the operation of the procedure for $n = 8$, showing also the sequence of divide and merge steps. The algorithm recursively divides the array down to 1-element subarrays. The combine steps merge pairs of 1-element subarrays to form sorted subarrays of length 2, merges those to form sorted subarrays of length 4, and merges those to form the final sorted subarray of length 8. If n is not an exact power of 2, then some divide steps create subarrays whose lengths differ by 1. (For example, when dividing a subarray of length 7, one subarray has length 4 and the other has length 3.) Regardless of the lengths of the two subarrays being merged, the time to merge a total of n items is $\Theta(n)$.

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call, you 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 of the same algorithm on smaller inputs. You can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm falls out from

the three steps of the basic method. As we did for insertion sort, let $T(n)$ be the worst-case running time on a problem of size n. If the problem size is small enough, say $n < n_0$ for some constant $n_0 > 0$, the straightforward solution takes constant time, which we write as $\Theta(1)$.[16] Suppose that the division of the problem yields a subproblems, each with size n/b, that is, $1/b$ the size of the original. For merge sort, both a and b are 2, but we'll see other divide-and-conquer algorithms in which $a \neq b$. It takes $T(n/b)$ time to solve one subproblem of size n/b, and so it takes $aT(n/b)$ time to solve all a of them. If it takes $D(n)$ 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

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 , \\ D(n) + aT(n/b) + C(n) & \text{otherwise} . \end{cases}$$

Chapter 4 shows how to solve common recurrences of this form.

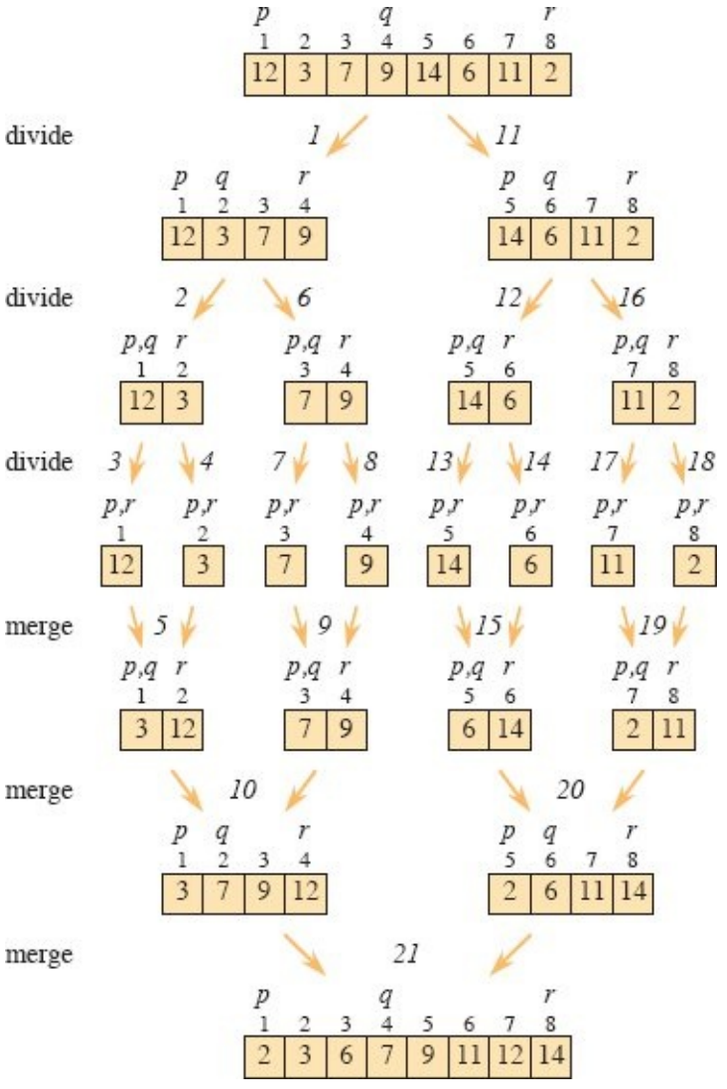

Figure 2.4 The operation of merge sort on the array A with length 8 that initially contains the sequence $\langle 12, 3, 7, 9, 14, 6, 11, 2\rangle$. The indices p, q, and r into each subarray appear above their values. Numbers in italics indicate the order in which the MERGE-SORT and MERGE procedures are called following the initial call of MERGE-SORT(A, 1, 8).

Sometimes, the n/b size of the divide step isn't an integer. For example, the MERGE-SORT procedure divides a problem of size n into subproblems of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Since the difference between $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ is at most 1, which for large n is much smaller than the effect of dividing n by 2, we'll squint a little and just call them both size $n/2$. As Chapter 4 will discuss, this simplification of ignoring floors and ceilings does not generally affect the order of growth of a solution to a divide-and-conquer recurrence.

Another convention we'll adopt is to omit a statement of the base cases of the recurrence, which we'll also discuss in more detail in Chapter 4. The reason is that the base cases are pretty much always $T(n) = \Theta(1)$ if $n < n_0$ for some constant $n_0 > 0$. That's because the running time of an algorithm on an input of constant size is constant. We save ourselves a lot of extra writing by adopting this convention.

Analysis of merge sort

Here's how to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

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

Conquer: Recursively solving two subproblems, each of size $n/2$, contributes $2T(n/2)$ to the running time (ignoring the floors and ceilings, as we discussed).

Combine: Since the MERGE procedure on an n-element subarray takes $\Theta(n)$ time, we have $C(n) = \Theta(n)$.

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, it is roughly proportional to n when n is large, and so merge sort's dividing and combining times together are $\Theta(n)$. Adding $\Theta(n)$ to the $2T(n/2)$ term from the conquer step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = 2T(n/2) + \Theta(n) . \tag{2.3}$$

Chapter 4 presents the "master theorem," which shows that $T(n) = \Theta(n \lg n)$.[17] Compared with insertion sort, whose worst-case running time is $\Theta(n^2)$, merge sort trades away a factor of n for a factor of $\lg n$. Because the logarithm function grows more slowly than any linear function, that's a good trade. For large enough inputs, merge sort, with its $\Theta(n \lg n)$ worst-case running time, outperforms insertion sort, whose worst-case running time is $\Theta(n^2)$.

We do not need the master theorem, however, to understand intuitively why the solution to recurrence (2.3) is $T(n) = \Theta(n \lg n)$. For simplicity, assume that n is an exact power of 2 and that the implicit base case is $n = 1$. Then recurrence (2.3) is essentially

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 , \end{cases} \tag{2.4}$$

where the constant $c_1 > 0$ represents the time required to solve a problem of size 1, and $c_2 > 0$ is the time per array element of the divide and combine steps.[18]

Figure 2.5 illustrates one way of figuring out the solution to recurrence (2.4). Part (a) of the figure shows $T(n)$, which part (b) expands into an equivalent tree representing the recurrence. The $c_2 n$ term denotes the cost of dividing and combining 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 for dividing and combining at each of the two nodes at the second level of recursion is $c_2 n/2$. Continue to expand 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_1. Part (d) shows the resulting ***recursion tree***.

Next, add the costs across each level of the tree. The top level has total cost $c_2 n$, the next level down has total cost $c_2(n/2) + c_2(n/2) = c_2 n$, the level after that has total cost $c_2(n/4) + c_2(n/4) + c_2(n/4) + c_2(n/4) = c_2 n$, and so on. Each level has twice as many nodes as the level above, but each node contributes only half the cost of a node from the level above. From one level to the next, doubling and halving cancel each other out, so that the cost across each level is the same: $c_2 n$. In general, the level that is i levels below the top has 2^i nodes, each contributing a cost of $c_2(n/2^i)$, so that the ith level below the top has total cost $2^i \cdot c_2(n/2^i) = c_2 n$. The bottom level has n nodes, each contributing a cost of c_1, for a total cost of $c_1 n$.

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 1 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 assume that the input size is an exact power of 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has 1 more level than a tree with 2^i leaves, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

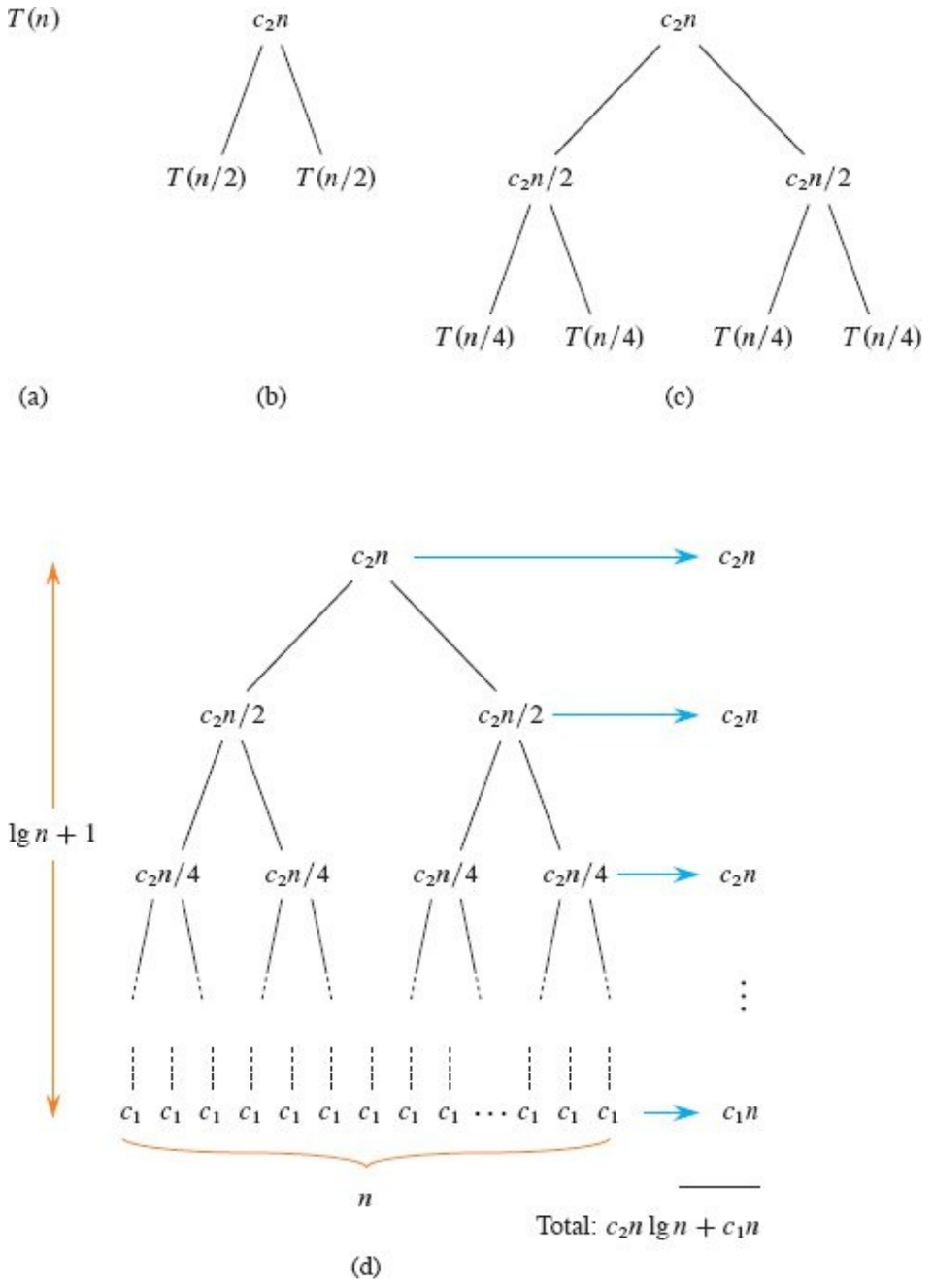

Figure 2.5 How to construct a recursion tree for the recurrence (2.4). Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels. Each level above the leaves contributes a total cost of $c_2 n$, and the leaf level contributes $c_1 n$. The total cost, therefore, is $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.

To compute the total cost represented by the recurrence (2.4), simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels. The levels above the leaves each cost $c_2 n$, and the leaf level costs $c_1 n$, for a total cost of $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.

Exercises

2.3-1

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

2.3-2

The test in line 1 of the MERGE-SORT procedure reads "**if** $p \geq r$" rather than "**if** $p \neq r$." If MERGE-SORT is called with $p > r$, then the subarray $A[p : r]$ is empty. Argue that as long as the initial call of MERGE-SORT$(A, 1, n)$ has $n \geq 1$, the test "**if** $p \neq r$" suffices to ensure that no recursive call has $p > r$.

2.3-3

State a loop invariant for the **while** loop of lines 12–18 of the MERGE procedure. Show how to use it, along with the **while** loops of lines 20–23 and 24–27, to prove that the MERGE procedure is correct.

2.3-4

Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

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

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

2.3-5

You can also think of insertion sort as a recursive algorithm. In order to sort $A[1 : n]$, recursively sort the subarray $A[1 : n - 1]$ and then insert $A[n]$ into the sorted subarray $A[1 : n - 1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

2.3-6

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

2.3-7

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 $A[1 : j - 1]$. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

2.3-8

Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

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 you 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. The procedure BUBBLESORT sorts array $A[1 : n]$.

```
BUBBLESORT(A, n)
1 for i = 1 to n – 1
2     for j = n downto i + 1
3         if A[j] < A[j – 1]
4             exchange A[j] with A[j – 1]
```

a. Let A' denote the array A after BUBBLESORT(A, n) is executed. To prove that

$$A'[1] \le A'[2] \le \cdots \le A'[n] . \tag{2.5}$$

In order to show that BUBBLESORT actually sorts, what else do you need to prove?

The next two parts prove inequality (2.5).

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 allows you to prove inequality (2.5). 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 with the running time of INSERTION-SORT?

2-3 Correctness of Horner's rule

You are given the coefficents $a_0, a_1, a_2, \ldots, a_n$ of a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n , \end{aligned}$$

and you want to evaluate this polynomial for a given value of x. ***Horner's rule*** says to evaluate the polynomial according to this parenthesization:

$$P(x) = a_0 + x\Big(a_1 + x\left(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots\right)\Big) .$$

The procedure HORNER implements Horner's rule to evaluate $P(x)$, given the coefficients $a_0, a_1, a_2, \ldots, a_n$ in an array $A[0 : n]$ and the value of x.

```
HORNER(A, n, x)
1   p = 0
2   for i = n downto 0
3       p = A[i] + x · p
4   return p
```

a. In terms of Θ-notation, what is the running time of this procedure?

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 with HORNER?

c. Consider the following loop invariant for the procedure HORNER:
At the start of each iteration of the **for** loop of lines 2–3,

$$p = \sum_{k=0}^{n-(i+1)} A[k + i + 1] \cdot x^k .$$

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, $p = \sum_{k=0}^{n} A[k] \cdot x^k$.

2-4 Inversions

Let $A[1 : n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

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

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

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

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

Chapter notes

In 1968, Knuth published the first of three volumes with the general title *The Art of Computer Programming* [259, 260, 261]. The first volume ushered in the modern study of computer algorithms with a focus on the analysis of running time. 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 [261] 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 subarrays 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 [200], who credits

P. Naur with the first article in this field. Gries attributes loop invariants to R. W. Floyd. The textbook by Mitchell [329] is a good reference on how to prove programs correct.

---

[1] If you're familiar with only Python, you can think of arrays as similar to Python lists.

[2] When the loop is a **for** loop, the loop-invariant check just prior to the first iteration occurs 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 i but before the first test of whether $i \le n$.

[3] In an **if-else** statement, we indent **else** at the same level as its matching **if**. The first executable line of an **else** clause appears on the same line as the keyword **else**. For multiway tests, we use **elseif** for tests after the first one. When it is the first line in an **else** clause, an **if** statement appears on the line following **else** so that you do not misconstrue it as **elseif**.

[4] Each pseudocode procedure in this book appears on one page so that you do not need to discern levels of indentation in pseudocode that is split across pages.

[5] Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat-until** loops, and its **for** loops operate differently from the **for** loops in this book. Think of the pseudocode line "**for** $i = 1$ **to** n" as equivalent to "for i in range(1, n+1)" in Python.

[6] In Python, the loop counter retains its value after the loop is exited, but the value it retains is the value it had during the final iteration of the **for** loop, rather than the value that exceeded the loop bound. That is because a Python **for** loop iterates through a list, which may contain nonnumeric values.

[7] If you're used to programming in Python, bear in mind that in this book, the subarray $A[i : j]$ includes the element $A[j]$. In Python, the last element of $A[i : j]$ is $A[j - 1]$. Python allows negative indices, which count from the back end of the list. This book does not use negative array indices.

[8] Python's tuple notation allows **return** statements to return multiple values without creating objects from a programmer-defined class.

[9] We assume that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations. For example, if array $A[1 : n]$ starts at memory address 1000 and each element occupies four bytes, then element $A[i]$ is at address $1000 + 4(i - 1)$. In general, computing the address in memory of a particular array element requires at most one subtraction (no subtraction for a 0-origin array), one multiplication (often implemented as a shift operation if the element size is an exact power of 2), and one addition. Furthermore, for code that iterates through the elements of an array in order, an optimizing compiler can generate the address of each element using just one addition, by adding the element size to the address of the preceding element.

[10] 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, in the RADIX-SORT procedure on page 213, one line reads "use a stable sort to sort array A on digit i," which, as we shall see, takes more than a constant amount of time. Also, although a statement that calls a subroutine takes only constant time, the subroutine itself, 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.

[11] 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.

[12] This procedure is the rare case that uses both 1-origin indexing (for array A) and 0-origin indexing (for arrays L and R). Using 0-origin indexing for L and R makes for a simpler loop invariant in Exercise 2.3-3.

[13] If you're wondering where the "+1" comes from, imagine that $r = p + 1$. Then the subarray $A[p : r]$ consists of two elements, and $r - p + 1 = 2$.

[14] Chapter 3 shows how to formally interpret equations containing Θ-notation.

[15] 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 Section 3.3. 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.

[16] If you're wondering where $\Theta(1)$ comes from, think of it this way. When we say that $n^2/100$ is $\Theta(n^2)$, we are ignoring the coefficient 1/100 of the factor n^2. Likewise, when we say that a constant c is $\Theta(1)$, we are ignoring the coefficient c of the factor 1 (which you can also think of as n^0).

[17] The notation lg n stands for $\log_2 n$, although the base of the logarithm doesn't matter here, but as computer scientists, we like logarithms base 2. Section 3.3 discusses other standard notation.

[18] It is unlikely that c_1 is exactly the time to solve problems of size 1 and that $c_2 n$ is exactly the time of the divide and combine steps. We'll look more closely at bounding recurrences in Chapter 4, where we'll be more careful about this kind of detail.