Chapter

2

Algorithm Discovery and Design

Chapter Introduction

After studying this chapter, you will be able to:

Explain the benefits of pseudocode over natural language or a programming language
Represent algorithms using pseudocode
Identify algorithm statements as sequential, conditional, or iterative
Define abstraction and top-down design, and explain their use in breaking down complex problems
Illustrate the operation of algorithms for:
Multiplication by repeated addition
Sequential search of a collection of values
Finding the maximum element in a collection
Finding a pattern string in a larger piece of text

2.1 Introduction

Chapter 1 introduced algorithms and algorithmic problem solving, two of the most fundamental concepts in computer science. Our introduction used examples drawn from everyday life, such as programming a DVR (Figure 1.1) and washing your hair (Figures 1.3 and 1.4). Although these are perfectly valid examples of algorithms, they are not of much interest to computer scientists. This chapter develops more fully the notions of algorithms and algorithmic problem solving and applies these ideas to problems that *are* of interest to computer scientists: searching lists, finding maxima and minima, and matching patterns.

2.2 Representing Algorithms

2.2.1 Pseudocode

Before presenting any algorithms, we must first make an important decision. How should we represent them? What notation should we use to express our algorithms so that they are clear, precise, and unambiguous?

One possibility is *natural language*, the language we speak and write in our everyday lives. (This could be English, Spanish, Arabic, Japanese, Swahili, or any language.) This is an obvious choice because it is the language with which we are most familiar. If we use natural language, then our algorithms would read much the same as a term paper or an essay. For example, when expressed in natural language, the addition algorithm in Figure 1.2 might look something like the paragraph shown in Figure 2.1.

Figure 2.1

The addition algorithm of Figure 1.2 expressed in natural language

Initially, set the value of the variable *carry* to 0 and the value of the variable *i* to 0. When these initializations have been completed, begin looping as long as the value of the variable *i* is less than or equal to (*m* – 1). First, add together the values of the two digits and and the current value of the carry digit to get the result called . Now check the value of to see whether it is greater than or equal to 10. If is greater than or equal to 10, then reset the value of *carry* to 1 and reduce the value of by 10; otherwise, set the value of *carry* to 0. When you are finished with that operation, add 1 to *i* and begin the loop all over again. When the loop has completed execution, set the leftmost digit of the result to the value of *carry* and print out the final result, which consists of the digits . After printing the result, the algorithm is finished, and it terminates.

Comparing Figure 1.2 with Figure 2.1 illustrates the problems of using natural language to represent algorithms. Natural language can be extremely verbose, causing the resulting algorithms to be rambling, unstructured, and hard to follow. (Imagine reading 5, 10, or even 100 pages of text like Figure 2.1.) An unstructured, "free-flowing" writing style might be wonderful for novels and essays, but it is horrible for algorithms. The lack of structure makes it difficult for the reader to locate specific sections of the algorithm because they are buried deep within the text. For example, about two-thirds of the way through Figure 2.1 is the phrase, ". . . and begin the loop all over again." To what part of the algorithm does this refer? Without any clues to guide us, such as indentation, line numbering, or highlighting, locating the beginning of that loop can be a daunting and time-consuming task. (For the record, the beginning of the loop corresponds to the sentence that starts, "When these initializations have been completed … ." It is certainly not easy to determine this from a casual reading of the text.)

A second problem is that natural language is too "rich" in interpretation and meaning. Natural language frequently relies on either context or a reader's experiences to give precise meaning to a word or phrase. This permits different readers to interpret the same sentence in totally different ways. This may be acceptable, even desirable, when writing poetry or fiction, but it is disastrous when

creating algorithms that must always execute in the same way and produce identical results. We can see an example of this problem in the sentence of Figure 2.1 that starts with "When you are finished with that operation . . . ." When we are finished with *which* operation? It is not at all clear from the text, and individuals might interpret the phrase *that operation* in different ways, producing radically different behavior. Similarly, the statement "Determine the shortest path between the source and destination" is ambiguous until we know the precise meaning of the phrase "shortest path." Does it mean shortest in terms of travel time, distance, or something else?

Because natural languages are not sufficiently precise to represent algorithms, we might be tempted to go to the other extreme. If we are ultimately going to execute our algorithm on a computer, why not immediately write it out as a computer program using a *high-level programming language* such as C++ or Java? If we adopt that approach, the addition algorithm of Figure 1.2 might start out looking like the program fragment shown in Figure 2.2.

Figure 2.2The beginning of the addition algorithm of Figure 1.2 expressed in a high-level programming language

As an algorithmic design language, this notation is also seriously flawed. During the initial phases of design, we should be thinking at a highly abstract level. However, using a formal programming language to express our design forces us to deal immediately with highly detailed language issues, such as punctuation, grammar, and syntax. For example, the algorithm in Figure 1.2 contains an operation that says, "Set the value of *carry* to 0." This is an easy statement to understand. However, when translated into a language like C++ or Java, that statement becomes

```
carry = 0;
```

Is this operation setting *carry* to 0 or asking if *carry* is equal to 0? Why does a semicolon appear at the end of the line? Would the statement

```
Carry = 0;
```

mean the same thing? Similarly, what is meant by the utterly cryptic statement on Line 4 of Figure 2.2: int [] a = new int [100];? These technical details clutter our thoughts and at this point in the solution process are totally out of place. When creating algorithms, a programmer should no more worry about semicolons and capitalization than a novelist should worry about typography and cover design when writing the first draft!

If the two extremes of natural languages and high-level programming languages are both less than ideal, what notation should we use? What is the best way to represent the solutions shown in this chapter and the rest of the book?

Most computer scientists use a notation called **pseudocode** to design and represent algorithms. This is a set of English-language constructs designed to more or less resemble statements in a programming language but that do not actually run on a computer. Pseudocode represents a compromise between the two extremes of natural and formal languages. It is simple, highly readable, and has virtually no grammatical rules. (In fact, pseudocode is sometimes jokingly referred to as "a programming language without the details.") However, because it contains only statements that have a well-defined structure, it is easier to visualize the organization of a pseudocode algorithm than one represented as long, rambling natural-language paragraphs. In addition, because pseudocode closely resembles many popular programming languages, the subsequent translation of the algorithm into a

computer program is relatively simple. The algorithms shown in Figures 1.1, 1.2, 1.3, 1.4, and Exercise 10 of Chapter 1 are all written in pseudocode.

In the following sections, we will introduce a set of popular and easyto- understand constructs for the three types of algorithmic operations introduced in Chapter 1: sequential, conditional, and iterative. Keep in mind, however, that pseudocode is *not* a formal language with rigidly standardized syntactic and semantic rules and regulations. On the contrary, it is an informal design notation used solely to express algorithms. If you do not like the constructs presented in the next two sections, feel free to modify them or select others that are more helpful to you. One of the nice features of pseudocode is that you can adapt it to your own personal way of thinking and problem solving.

2.2.2 Sequential Operations

Our pseudocode must include instructions to carry out the three basic *sequential operations* called computation, input, and output.

The instruction for performing a **computation** and saving the result looks like the following. (Words and phrases inside quotation marks represent specific elements that you must insert when writing an algorithm.)

Set the value of "variable" to "arithmetic expression"

This operation evaluates the "arithmetic expression," gets a result, and stores that result in the "variable." A **variable** is simply a named storage location that can hold a data value. A variable is often compared with a mailbox into which you can place a value and from which you can retrieve a value. Let's look at an example.

Set the value of *carry* to 0

First, evaluate the arithmetic expression, which in this case is the constant value 0. Then store that result in the variable called *carry*. If *carry* had a previous value, say 1, that value will be discarded and replaced by 0. If *carry* did not yet have a value, it now does—the value 0. You can visualize the result of this operation as follows:

Here is another example:

Set the value of *Area* to (π r 2)

Assuming that the variable *r* has been given a value by a previous instruction in the algorithm, this statement evaluates the arithmetic expression π r 2 to produce a numerical result. This result is then stored in the variable called *Area*. If *r*does not have a value, an error condition occurs because this instruction is not effectively computable, and it cannot be completed.

We can see additional examples of computational operations in Steps 4, 6, and 7 of the addition algorithm of Figure 1.2:

1. Step 4
   Add the two digits a i and b i to the current value of *carry* to get c i
2. Step 6
   Add 1 to *i*, effectively moving one column to the left

3. Step 7
Set c m to the value of *carry*

Note that these three steps are not written in exactly the format just described. If we had used that notation, they would have looked like this:

1. Step 4
Set the value of c i to (a i + b i +carry)

2. Step 6
Set the value of *i* to (*i* + 1)

3. Step 7
Set the value of c m to *carry*

However, in pseudocode, it doesn't matter exactly how you choose to write your instructions as long as the intent is clear, effectively computable, and unambiguous. At this point in the design of a solution, we do not care about the minor linguistic differences between

Add *a* and *b* to get *c*

and

Set the value of *c* to (*a* + *b*)

Remember that pseudocode is not a precise set of notational rules to be memorized and rigidly followed. It is a flexible notation that can be adjusted to fit your own view about how best to express ideas and algorithms.

When writing arithmetic expressions, you may assume that the computing agent executing your algorithm has all the capabilities of a typical multifunction calculator. Therefore, it "knows" how to do all basic arithmetic operations such as +, −, ×, ÷, √, absolute value, sine, cosine, and tangent. It also knows the value of important constants such as π.

The remaining two sequential operations enable our computing agent to communicate with "the outside world," which means everything other than the computing agent itself:

Input operations provide the computing agent with data values from the outside world that it may then use in later instructions. **Output** operations send results from the computing agent to the outside world. When the computing agent is a computer, communications with the outside world are done via the input/output equipment available on a typical computer, tablet, or smartphone, such as a physical keyboard, virtual keypad, screen, mouse, printer, hard drive, camera, or touch screen. However, when designing algorithms, we generally do not concern ourselves with the technical specifications of a particular device. At this point in the design process, we care only that data is provided to us when we request it, and that results are issued for presentation.

Our pseudocode instructions for input and output are expressed as follows:

Input: Get values for "variable", "variable", . . .

Output: Print the values of "variable", "variable", . . .

For example,

Get a value for *r*, the radius of the circle

When the algorithm reaches this input operation, it waits until someone or something provides it with a value for the variable *r*. (In a computer, this may be done by entering a value at the keyboard.) When the algorithm has received and stored a value for *r*, it continues on to the next instruction.

Here is an example of an output operation:

Print the value of *Area*

Assuming that the algorithm has already computed the area of the circle, this instruction says to display that value to the outside world. This display may be viewed on a screen (computer, tablet, smartphone) or printed on paper by a printer.

Sometimes we use an output instruction to display a message in place of the desired results. If, for example, the computing agent cannot complete a computation because of an error condition, we might have it execute something like the following operation. (We will use 'single quotation marks' to enclose messages so as to distinguish them from such pseudocode constructs as "variable" and "arithmetic expression," which are enclosed in double quotation marks.)

Print the message 'Sorry, no answers could be computed.'

Using these three sequential operations—computation, input, and output—we can now write some simple but useful algorithms. Figure 2.3 presents an algorithm to compute the average miles per gallon on a trip, when given as input the number of gallons used and the starting and ending mileage readings on the odometer.

Figure 2.3 Algorithm for computing average miles per gallon (version 1)

2.2.3 Conditional and Iterative Operations

The average miles per gallon algorithm in Figure 2.3 performs a set of operations once and then stops. It cannot select among alternative operations or perform a block of instructions more than once. A purely **sequential algorithm** of the type shown in Figure 2.3 is sometimes termed a *straight-line algorithm* because it executes its instructions in a straight line from top to bottom and then stops. Unfortunately, virtually all real-world problems are not straight-line in nature. They involve nonsequential operations such as branching and repetition.

To allow us to address these more interesting problems, our pseudocode needs two additional statements to implement conditional and **iterative operations**. Together, these two types of operations are called **control operations**; they allow us to alter the normal sequential flow of control in an algorithm. As we saw in Chapter 1, control operations are an essential part of all but the very simplest of algorithms.

Practice Problems

Write pseudocode versions of the following:

1. An algorithm that gets three data values *x*, *y*, and *z* as input and outputs the average of those three values.

   Answer

2. An algorithm that gets the radius *r* of a circle as input. Its output is both the circumference and the area of a circle of radius *r*.

   Answer

3. An algorithm that gets the amount of electricity used in kilowatt-hours and the cost of electricity per kilowatt-hour. Its output is the total amount of the electric bill, including an 8% sales tax.

   Answer

4. An algorithm that inputs your current credit card balance, the total dollar amount of new purchases, and the total dollar amount of all payments. The algorithm computes the new balance, which includes a 12% interest charge on any unpaid balance.

   Answer

5. An algorithm that is given the length and width, in feet, of a rectangular carpet and determines its total cost given that the material cost is $23 per square yard.

   Answer

6. An algorithm that is given three numbers corresponding to the number of times a race car driver has finished first, second, and third. The algorithm computes and displays how many points that driver has earned given 5 points for a first, 3 points for a second, and 1 point for a third place finish.

   Answer

Conditional statements are the "question-asking" operations of an algorithm. They allow an algorithm to ask a yes/no question and select the next operation to perform on the basis of the answer to that question. There are a number of ways to phrase a question, but the most common conditional statement is the *if/then/else* statement, which has the following format:

The meaning of this statement is as follows:

1. Evaluate the true/false condition on the first line to determine whether it is true or false.
2. If the condition is true, then do the first set of algorithmic operations and skip the second set entirely.
3. If the condition is false, then skip the first set of operations and do the second set.
4. Once the appropriate set of operations has been completed, continue executing the algorithm with the operation that follows the if/then/else instruction.

Figure 2.4 is a visual model of the execution of the if/then/else statement. We evaluate the condition shown in the diamond. If the condition is true, we execute the sequence of operations labeled T1, T2, T3, … . If the condition is false, we execute the sequence labeled F1, F2, F3, … . In either case, however, execution continues with statement S, which is the one that immediately follows the if/then/else statement.

Figure 2.4The if/then/else pseudocode statement

Basically, the if/then/else statement allows you to select exactly one of two alternatives—either/or, this or that. We saw an example of this statement in Step 5 of the addition algorithm shown in Figure 1.2. (The statement has been reformatted slightly to highlight the two alternatives clearly, but it has not been changed.)

The condition (c i ≥10) can be only true or false. If it is true, then there is a carry into the next column, and we must do the first set of instructions—subtracting 10 from c i and setting *carry* to 1. If the condition is false, then there is no carry—we skip over these two operations and perform the second block of operations, which simply sets the value of *carry* to 0.

Figure 2.5 shows another example of the if/then/else statement. It extends the miles per gallon algorithm of Figure 2.3 to include a second line of output stating whether you are getting good gas mileage. Good gas mileage is defined as a value for average miles per gallon strictly greater than 25.0 miles per gallon.

Figure 2.5Second version of the average miles per gallon algorithm

The last algorithmic statement to be introduced allows us to implement a *loop*—the repetition of a block of instructions. The real power of a computer comes not from doing a calculation once but from doing it many, many times. If, for example, we need to compute a single value of average miles per gallon, it would be foolish to convert an algorithm like Figure 2.5 into a computer program and execute it on a computer—it would be far faster to use a calculator, which could complete the job in a few seconds. However, if we need to do the same computation 1 million times, the power of a computer to repetitively execute a block of statements becomes quite apparent. If each computation of average miles per gallon takes 5 seconds on a hand calculator, then 1 million of them would require about 2 months, not allowing for such luxuries as sleeping and eating. Once the algorithm is developed and the program written, a computer could carry out that same task in a fraction of a second!

The first algorithmic statement that we will use to express the idea of **iteration**, also called *looping*, is the *while*statement:

This instruction initially evaluates the “true/false condition”—called the **continuation condition**—to determine if it is true or false. If the condition is true, all operations from Step *i* to Step *j*, inclusive, are executed. This block of operations is called the **loop body**. (Operations within the loop body should be indented so that it is clear to the reader of the algorithm which operations belong inside the loop.) When the entire loop body has finished executing, the algorithm again evaluates the continuation

condition. If it is still true, then the algorithm executes the entire loop body, statements *i* through *j*, again. This looping process continues until the continuation condition evaluates to false, at which point execution of the loop body terminates and the algorithm proceeds to the statement immediately following the loop—Step *j* + 1 in the above pseudocode. If for some reason the continuation condition never becomes false, then we have violated one of the fundamental properties of an algorithm, and we have the error, first mentioned in Chapter 1, called an *infinite loop*.

Figure 2.6 is a visual model of the execution of a while loop. The algorithm first evaluates the continuation condition inside the diamond-shaped symbol. If it is true, then it executes the sequence of operations labeled S1, S2, S3, …, which are the operations of the loop body. Then the algorithm returns to the top of the loop and reevaluates the condition. If the condition is false, then the loop has ended, and the algorithm continues executing with the statement after the loop, the one labeled S n in Figure 2.6.

Figure 2.6Execution of the while loop

Here is a simple example of a loop:

Step 1 initializes *count* to 1, the next operation determines that (*count* ≤ 100), and then the loop body is executed, which in this case includes the three statements in Steps 3–5. Those statements compute the value of *count* squared (Step 3) and print the value of both *count* and *square* (Step 4). The last operation inside the loop body (Step 5) adds 1 to *count* so that it now has the value 2. At the end of the loop, the algorithm must determine whether it should be executed again. Because *count* is 2, the continuation condition (*count* ≤ 100) is still true, and the algorithm must perform the loop body again. Looking at the entire loop, we can see that it will execute 100 times, producing the following output, which is a table of numbers and their squares from 1 to 100.

1	1
2	4
3	9
•	•
•	•
•	•
100	10,000

At the end of the 100th pass through the loop, the value of *count* is incremented in Step 5 to 101. When the continuation condition is evaluated, it is false (because 101 is not less than or equal to 100), and the loop terminates.

We can see additional examples of loop structures in Steps 3 through 6 of Figure 1.2 and in Steps 3 through 6 of Figure 1.3. Another example is shown in Figure 2.7, which is yet another variation of the average miles per gallon algorithm of Figures 2.3 and 2.5. In this modification, after finishing one computation, the algorithm asks the user whether to repeat this calculation again. It waits until it gets

a Yes or No response and repeats the entire loop body until the *response* provided by the user is No. (Note that the algorithm must initialize the value of response to Yes because the very first thing that the loop does is test the value of this quantity.)

Figure 2.7 Third version of the average miles per gallon algorithm

There are many variations of this particular looping construct in addition to the while statement just described. For example, it is common to omit the line numbers from algorithms and simply execute them in order, from top to bottom. In that case, we could use an "End of the loop" construct (or something similar) to mark the end of the loop rather than explicitly stating which steps are contained in the loop body. Using this approach, our loops would be written something like this:

In this case, the loop body is delimited not by explicit step numbers but by the two lines that read, "While . . ." and "End of the loop".

The type of loop just described is called a *pretest loop* because the continuation condition is tested at the *beginning* of each pass through the loop, and therefore it is possible for the loop body never to be executed. (This would happen if the continuation condition were *initially* false.) Sometimes this can be inconvenient, as we see in Figure 2.7. In that algorithm, the value of the variable called *response* is tested for the first time long before we ask the user if he or she wants to solve the problem again. Therefore, we had to give *response* a "dummy" value of Yes so that the test would be meaningful when the loop was initially entered.

A useful variation of the looping structure is called a *posttest* loop, which also uses a true/false continuation condition to control execution of the loop. However, now the test is done at the *end* of the loop body, not the beginning. The loop is typically expressed using the *do/while* statement, which is usually written as follows:

This type of iteration performs all the algorithmic operations contained in the loop body before it evaluates the true/false condition specified at the end of the loop. If this condition is false, the loop is terminated and execution continues with the operation following the loop. If the condition is true, then the entire loop body is executed again. Note that in the do/while variation, the loop body is always executed at least once, whereas the while loop can execute 0, 1, or more times. Figure 2.8 diagrams the execution of the posttest do/while looping structure.

Figure 2.8 Execution of the do/while posttest loop

Figure 2.9 summarizes the algorithmic operations introduced in this section. These represent the **primitive operations** of our computing agent. These are the instructions that we assume our computing agent understands and is capable of executing without further explanation or simplification. In the next section, we will use these operations to design algorithms that solve some interesting and important problems.

Figure 2.9 Summary of pseudocode language instructions

From Little Primitives Mighty Algorithms Grow

Although the set of algorithmic primitives shown in Figure 2.9 might seem quite puny, it is anything but! In fact, an important theorem in theoretical computer science proves that the operations shown in Figure 2.9 are sufficient to represent *any* valid algorithm. No matter how complicated it might be, if a problem can be solved algorithmically, it can be expressed using only the sequential, conditional, and iterative operations just discussed. This includes not only the simple addition algorithm of Figure 1.2but also the exceedingly complex algorithms needed to operate the International Space Station, manage billions of Facebook accounts, and implement all the Internal Revenue Service's tax rules and regulations.

In many ways, building algorithms is akin to constructing essays or novels using only the 26 letters of the English alphabet, plus a few punctuation symbols. Expressive power does not always come from having a huge set of primitives. It can also arise from a small number of simple building blocks combined in interesting and complex ways. This is the real secret of building algorithms.

2.3Examples of Algorithmic Problem Solving

2.3.1Example 1: Go Forth and Multiply

Our first example of algorithmic problem solving addresses a problem originally posed in Chapter 1 (Exercise 12). That problem asked you to implement an algorithm to multiply two numbers using repeated addition. This problem can be formally expressed as follows:

Given two nonnegative integer values, $a \geq 0$, $b \geq 0$, compute and output the product ($a \times b$) using the technique of repeated addition. That is, determine the value of the sum $a + a + a + \ldots + a$ ($b$times).

Obviously, we need to create a loop that executes exactly b times, with each execution of the loop adding the value of a to a running total. These operations will not make any sense (that is, they will not be effectively computable) until we have explicit values for a and b. So one of the first operations in our algorithm must be to input these two values.

Get values for a and b

Practice Problems

1. Write an if/then/else statement that sets the variable y to the value 1 if $x \geq 0$. If $x < 0$, then the statement should set y to the value 2. (Assume x already has a value.)

   Answer

2. Write an algorithm that gets as input three data values x, y, and z and outputs the average of these values if the value of x is positive. If the value of x is either 0 or negative, your algorithm should not compute the average but should print the error message 'Bad Data' instead.

   Answer

3. Write an algorithm that gets as input your current credit card balance, the total dollar amount of new purchases, and the total dollar amount of all payments. The algorithm computes the new balance, which this time includes an 8% interest charge on any unpaid balance below $100, 12% interest on any unpaid balance between $100 and $500, inclusive, and 16% on any unpaid balance above $500.

Answer

4. Write an algorithm that gets as input a single nonzero data value x and outputs the three values , sin x, and $1/x$. This process is repeated until the input value for x is equal to 999, at which time the algorithm terminates.
Answer

5. Write an algorithm that inputs the length and width, in feet, of a rectangular carpet and the price of the carpet in dollars per square yard. It then determines if you can afford to purchase this carpet, given that your total budget for carpeting is $500.

Answer

6. Add the following feature to the algorithm created in the previous Practice Problem: If the cost of the carpet is less than or equal to $250, output a message that this is a particularly good deal.

Answer

Add the following statement to the end of the solution, just before the Stop statement:

(Assume that we have stored the cost of the carpet in the variable *cost*.)

7. Add the following feature to the algorithm created in Practice Problem 4 above. Assume the input for the data value x can be any value, including 0. Since the value $1/x$ cannot be computed if , you will have to check for this condition and output an error message saying that you are unable to compute the value $1/x$.
Answer

To create a loop that executes exactly b times, we create a counter, let's call it *count*, initialized to 0 and incremented by (increased by) 1 after each pass through the loop. This means that when we have completed the loop once, the value of *count* is 1; when we have completed the loop twice, the value of *count* is 2; and so forth. Because we want to stop when we have completed the loop b times, we want to stop when . Therefore, the condition for continuing execution of the loop is ($count < b$). Putting all these pieces together produces the following algorithmic structure, which is a loop that executes exactly b times as the variable *count* ranges from 0 up to ($b - 1$).

The purpose of the loop body is to add the value of a to a running total, which we will call *product*. We express that operation in the following manner:

Set the value of *product* to ($product + a$)

This statement says the new value of *product* is to be reset to the current value of *product*plus a.

What is the current value of *product* the first time this operation is encountered? Unless we initialize it, it has no value, and this operation is not effectively computable. Before starting the loop, we must be sure to include the following step:

Set the value of *product* to 0

Now our solution is starting to take shape. Here is what we have developed so far:

There are only a few minor "tweaks" left to make this a correct solution to our problem.

When the while loop completes, we have computed the desired result, namely ($a \times b$), and it is stored in *product*. However, we have not displayed that result, and as it stands, this algorithm produces no output. Remember from Chapter 1 that one of the fundamental characteristics of an algorithm is that it produces an observable result. In this case, the desired result is the final value of *product*, which we can display using our output primitive:

Print the value of *product*

The original statement of the problem said that the two inputs *a* and *b* must satisfy the following conditions: a ≥ 0 and b ≥ 0. The previous algorithm works for positive values of *a*and *b*, but what happens when either or ? Does it still function correctly?
If , there is no problem. If you look at the while loop, you see that it continues executing so long as (*count* < *b*). The variable *count* is initialized to 0. If the input variable *b*also has the value 0, then the test (0 < 0) is initially false, and the loop is *never* executed. The variable *product* keeps its initial value of 0, and that is the output that is displayed, which is the correct answer.
Now let's look at what happens when and *b* is any nonzero value, say 5,386. Of course we know immediately that the correct answer to 0 × 5,386 is 0, but our algorithm does not. Instead, the loop will execute 5,386 times, the value of *b*, each time adding the value of *a*, which is 0, to *product*. Because adding 0 to anything has no effect, *product* remains at 0, and that is the output that will be displayed. In this case, we do get the right answer, and our algorithm does work correctly. However, it gets that correct answer only after doing 5,386 unnecessary and time-wasting repetitions of the loop.
In Chapter 1, we stated that it is not only algorithmic correctness we are after but efficiency and elegance as well. The algorithms designed and implemented by computer scientists are intended to solve important realworld problems, and they must accomplish that task in a correct and reasonably efficient manner. Otherwise they are not of much use to their intended audience.

In this case, we can eliminate all of those needless repetitions of the loop by using our if/then/else conditional primitive. Right at the start of the algorithm, we ask if either *a* or *b* is equal to 0. If the answer is yes, we can immediately set the final result to 0 without requiring any further computations:

We will have much more to say about the critically important concepts of algorithmic efficiency and elegance in Chapter 3.

This completes the development of our multiplication algorithm, and the finished solution is shown in Figure 2.10.

Figure 2.10Algorithm for multiplication of nonnegative values via repeated addition

Practice Problems

1. Manually work through the algorithm in Figure 2.10 using the input values , . After each completed pass through the loop, write down the current value of the four variables *a*, *b*, *count*, and *product*.
Answer

Initial values

After pass 1

After pass 2

After pass 3

After pass 4

2. Trace the execution of the algorithm in Figure 2.10 using the "special" input values and . Does the algorithm produce the result you expect?
Answer

Yes, the algorithm still works correctly. On Line 1, we input the two values , . On Line 2, the Boolean expression is true, so we set and skip the entire else clause. The next line executed is "print the value of product" which outputs a 0, the correct answer to the problem 0 × 0.

3. Describe exactly what would be output by the algorithm in Figure 2.10 for each of the following two cases, and state whether that output is or is not correct. (*Note*: Because one of the two inputs is negative, these values violate the basic conditions of the problem.)

case 1: ,

case 2: ,

4. Answer
5. case 1 (,):
6. The value of *product* will be −8,

which is correct.

7. case 2 (,):
8. The value of *product* will be 0 (the

while loop does not execute at all), which is incorrect.

9. If the algorithm of Figure 2.10 produced the wrong answer for either case 1 or case 2 of Practice Problem 3, explain exactly how you could fix the algorithm so it works correctly and produces the correct answer.

Answer

The original algorithm fails when $b < 0$ because *count* is never less than b. If $b < 0$, change the value of b to $-b$, but set the value of a variable called *bnegative* to YES to remember that b was negative. After the product is computed, the sign of *product* will be incorrect if b was negative, so change the sign. Here is a pseudocode version that works for all integer values of a and b:

10. Explain why the multiplication algorithm shown in Figure 2.10 is or is not an efficient way to do multiplication. Justify and explain your answer.

Answer

It is a highly inefficient algorithm because of the number of steps required to find the answer. For example, to add 234 + 567, we will have to repeat the addition of the value 234 to a running sum 567 separate times. When we do multiplication the "traditional" way that we learned in grade school, we have to carry out far fewer operations. (We will learn much more about how to evaluate the efficiency of algorithms in Chapter 3.)

11. Modify the algorithm in Figure 2.10 so it examines the two inputs a and b and if either one is greater than 10,000, it displays a message saying that for large numbers like this we should use a different, and more efficient, multiplication algorithm. It then terminates without computing a result.

Answer

This first example needed only two integer values, *a* and *b*, as input. That is a bit unrealistic, as most interesting computational problems deal not with a few numbers but with huge collections of data, such as long lists of names or large sets of experimental data. In the following sections, we will show examples of the types of processing—searching, reordering, comparing—often done on huge collections of information.

2.3.2 Example 2: Looking, Looking, Looking

Finding a solution to a given problem is called **algorithm discovery**, and it is the most challenging and creative part of the problem-solving process. We developed an algorithm for a fairly simple problem (multiplication by repeated addition) in Example 1. Discovering a correct and efficient algorithm to solve a complicated problem can be difficult and can involve equal parts of intelligence, hard work, past experience, technical skill, and plain good luck. In the remaining examples, we will develop solutions to a range of problems to give you more experience in working with algorithms. Studying these examples, together with lots of practice, is by far the best way to learn creative problem solving, just as experience and practice are the best ways to learn how to write essays, hit a golf ball, or repair cars.

The next problem we address involves finding a person's name given his or her telephone number, an application often referred to as *reverse telephone lookup*. This is the type of important but rather menial repetitive task so well suited to computerization. (Apple has a half-dozen reverse telephone lookup apps in its App Store.)

This algorithm could be used, for example, to implement *Caller ID*, in which the caller's name (if it is found) is shown on a screen so you can decide whether or not to answer the phone. It can also be used with missed calls to determine whether the caller is someone with whom you actually wish to speak. Since there are more than 350 million listed phone numbers in the United States, reverse telephone lookup can only be implemented using computer-based search techniques.

Assume that we have a list of 10,000 telephone numbers (rather than hundreds of millions) that we represent symbolically as , , , ..., , along with the 10,000 names of the individuals associated with each specific phone number, denoted as , , , ..., . That is, is the name of the person who "owns" the telephone with the number , and so forth. On this first attempt to build an algorithm, let's assume that the 10,000 telephone numbers are not necessarily in numerical order. (In Chapter 3, we will modify the problem to search a list of phone numbers sorted into ascending order.) Essentially what we have described is randomly ordered pairs of number/name lists with the following structure:

Our algorithm should allow us to input a specific telephone number, which we will symbolically denote as *NUMBER*. The algorithm will then search to see if *NUMBER* matches any of the 10,000 numbers contained in our reverse directory. If *NUMBER* matches , where *j* is some value between 1 and 10,000, then the output of our algorithm will be the name of the person associated with that number: the value . If *NUMBER* is not in our reverse directory, then the output of our algorithm will be the message 'I am sorry but this number is not in our directory.' This type of lookup algorithm has many additional uses. For example, it could be used to locate the zip code of a particular city, the seat number of a specific airline passenger, or the room number of a hotel guest.

Because the numbers in our directory are not in numerical order, there is no clever way to speed up the search. With a randomly ordered collection, there is no method more efficient than starting at the beginning and looking at each number in the list, one at a time, until we either find the one we are looking for or we come to the end of the list. This rather simple and straightforward technique is called *sequential search*, and it is the standard algorithm for searching an *unordered* list of values. For example, this is how we would search a bookshelf for a book with a particular title if the books were sorted by the author's name instead of by title. It is also the way that we would search a shuffled deck of cards trying to locate one particular card. A first attempt at designing a sequential search algorithm to solve our search problem might look something like Figure 2.11.

Figure 2.11 First attempt at designing a sequential search algorithm

The solution shown in Figure 2.11 is extremely long. At 66 lines per page, it would require about 150 pages to write out the 10,002 steps in the completed solution. It would also be unnecessarily slow. If we are lucky enough to find *NUMBER* in the very first position of the list, , then we get the answer almost immediately. However, the algorithm does not stop at that point. Even though it has already found the correct answer, it foolishly asks 9,999 more questions looking for *NUMBER* in positions , . . ., . Of course, humans have enough common sense to know that when they find the answer they are searching for, they can stop. However, we cannot assume common sense in a computer system. On the contrary, a computer will mechanically execute the entire algorithm from the first step to the last.

Not only is the algorithm excessively long and highly inefficient, it is also wrong. If the desired *NUMBER* is not in the list, this algorithm simply stops (at Step 10,002) rather than providing the desired result, a message that the number you requested could not be found. An algorithm is deemed correct only when it produces the correct result for *all* possible cases.

The problem with this first attempt is that it does not use the powerful algorithmic concept of *iteration*. Instead of writing an instruction 10,000 separate times, it is far better to write it only once and indicate that it is to be repetitively *executed* 10,000 times, or however many times it takes to obtain the answer. As you learned in the previous section, much of the power of a computer comes from being able to perform a *loop*—the repetitive execution of a block of statements a large number of times. Virtually every algorithm developed in this text contains at least one loop and most contain many. (This is the difference between the two shampooing algorithms shown in Figures 1.3 and 1.4. The algorithm in the former contains a loop; that in the latter does not.)

The algorithm in Figure 2.12 shows how we might write a loop to implement the sequential search technique. It uses a variable called *i* as an *index*, or *pointer*, into the list of all numbers. That is, refers to the *i*th number in the list. The algorithm then repeatedly executes a group of statements using different values of *i*. The variable *i* can be thought of as a "moving finger" scanning the list of telephone numbers and pointing to the one on which the algorithm is currently working.

Figure 2.12 Second attempt at designing a sequential search algorithm

The first time through the loop, the value of the index *i* is 1, so the algorithm checks (in Step 4) to see whether *NUMBER* is equal to , the first one on the list. If it is, then the algorithm writes out the result and sets the variable *Found* to *YES*, which causes the loop in Steps 4 through 7 to terminate. If is not the desired *NUMBER*, then *i* is incremented by 1 (in Step 7) so that it now has the value 2, and the loop is executed again. The algorithm now checks to see whether *NUMBER* is equal to , the second number on the list. In this way, the algorithm uses the single conditional statement "If *NUMBER* is equal to the *i*th number on the list . . ." to check up to 10,000 different values. It executes that one line over and over, each time with a different value of *i*. This is the advantage of using iteration.
However, the attempt shown in Figure 2.12 is not yet a complete and correct algorithm because it still does not work correctly when the desired *NUMBER* does not appear anywhere in our reverse directory. This final problem can be solved by terminating the loop either when the desired phone number is found or when we reach the end of the list. The algorithm can determine exactly what happened by checking the value of *Found* when the loop terminates. If the value of *Found* is NO, then the loop terminated because the index *i*exceeded 10,000, and we searched the entire list without finding the desired *NUMBER*. The algorithm should then produce an appropriate message.

An iterative solution to the sequential search algorithm that incorporates this feature is shown in Figure 2.13. The sequential search algorithm shown in Figure 2.13 is a correct solution to our reverse telephone lookup problem. It meets all the requirements listed in Section 1.3.1: It is well ordered, each of the operations is clearly defined and effectively computable, and it is certain to halt with the desired result after a finite number of operations. (In Exercise 12 at the end of this chapter, you will develop a formal argument that proves that this algorithm will always halt.) Furthermore, this algorithm requires only 10 steps to write out fully, rather than the 10,002 steps of the first attempt in Figure 2.11. As you can see, not all algorithms are created equal.

Figure 2.13The sequential search algorithm

Looking at the algorithm in Figure 2.13, our first thought might be that this is not at all how people would manually search a list of telephone numbers looking for one specific value. Humans would never turn to page 1, column 1, and start scanning all numbers beginning with (000) 000-0001 (which is not actually a valid telephone number according to the North American Numbering Plan that covers the United States, Canada, and many Caribbean islands). In all likelihood, a communications company located in New York City would not be satisfied with the performance of the sequential search algorithm of Figure 2.13 when applied to the 20 million or so phones in its city.

But because our reverse directory was not sorted into numerical order, we really had no choice in the design of our search algorithm. However, in real life we can do much better than sequential search, because these types of directories *are* sorted numerically, and we can exploit this fact during the search process. For example, we know that the digit 5 is about halfway through the set of decimal digits 0–9. So when looking for the owner of phone number (555) 123-4567 in a sorted reverse directory, we could start our search somewhere in the middle rather than on the first page. We then see exactly where we are by looking at the first digit of the phone numbers on the current page and then move forward or backward toward numbers beginning with 5. This approach allows us to find the desired telephone number much more quickly than searching the numbers sequentially from the beginning of the list.

This use of different search techniques points out a very important concept in the design of algorithms:

The selection of an algorithm to solve a problem is greatly influenced by the way the input data for that problem is organized.

An algorithm is a method for processing some data to produce a result, and the way the data is organized has an enormous influence both on the algorithm we select and on how speedily that algorithm can produce the desired result.

In Chapter 3, we will expand on the concept of the efficiency and quality of algorithms, and we will present an algorithm for searching *sorted* reverse directories that is far superior to the one shown in Figure 2.13.

2.3.3 Example 3: Big, Bigger, Biggest

The third algorithm we will develop is similar to the sequential search in Figure 2.13 in that it also searches a list of values. However, this time the algorithm will search not for a particular value supplied by the user but for the numerically largest value in a list of numbers. This type of "find largest" algorithm could be used to answer a number of important questions. (With only a single trivial change, the same algorithm also finds the smallest value, so a better name for it might be "find extreme values.") For example, given a list of examinations, which student received the highest (or lowest) score? Given a list of annual salaries, which employee earns the most (or least) money? Given a list of grocery prices from different stores, where should I shop to find the lowest price? All these questions could be answered by executing this type of algorithm.

In addition to being important in its own right, such an algorithm can also be used as a "building block" for the construction of solutions to other problems. For example, the Find Largest algorithm that we will develop could be used to implement a *sorting algorithm* that puts an unordered list of numbers into ascending order. (Find and remove the largest item in list A and move it to the last position of list B. Now repeat these operations, each time moving the largest remaining number in list A to the last unfilled slot of list B. We will develop and write this algorithm in Chapter 3.)

The use of a "building-block" component is a very important concept in computer science. The examples in this chapter might lead you to believe that every algorithm you write must be built from only the most elementary and basic of primitives—the sequential, conditional, and iterative operations shown in Figure 2.9. However, once an algorithm has been developed, it may itself be used in the construction of other, more complex algorithms, just as we will use Find Largest in the design of a sorting algorithm. This is similar to what a builder does when constructing a home from prefabricated units rather than bricks and boards. Our problem-solving task need not always begin at the beginning but can instead build on ideas and results that have come before. Every algorithm that we create becomes, in a sense, a primitive operation of our computing agent and can be used as part of

the solution to other problems. That is why a collection of useful, prewritten algorithms, called a **library**, is such an important tool in the design and development of algorithms.

Formally, the problem we will be solving in this section is defined as follows:

Given a value $n \geq 1$ and a list containing exactly n unique numbers called , , …, , find and print both the largest value in the list and the position in the list where that largest value occurred.
For example, if our list contained the five values

19, 41, 12, 63, 22
then our algorithm should locate the largest value, 63, and print that value together with the fact that it occurred in the fourth position of the list. (*Note*: Our definition of the problem states that all numbers in the list are unique, so there can be only a single occurrence of the largest number. Exercise 15 at the end of the chapter asks how our algorithm would behave if the numbers in the list were not unique and the largest number could occur two or more times.)

When faced with a problem statement like the one just given, how do we go about creating a solution? What strategies can we employ to discover a correct and efficient answer to the problem? One way to begin is to ask ourselves how the same problem might be solved by hand. If we can understand and explain how we would approach the problem manually, we might be able to express that manual solution as a formal algorithm.

For example, suppose we were given a pile of papers, each of which contains a single number, and were asked to locate the largest number in the pile. (The following diagrams assume the papers contain the five values 19, 41, 12, 63, and 22.)

We might start off by saying that the first number in the pile (the top one) is the largest one that we have seen so far, and then putting it to the side where we are keeping the largest value.

Now we compare the top number in the pile with the one that we have called the largest one so far. In this case, the top number in the pile, 41, is larger than our current largest, 19, so we make it the new largest. To do this, we throw the value 19 into the wastebasket (or, better, into the recycle bin) and put the number 41 to the side because it is the largest value encountered so far.

We now repeat this comparison operation, asking whether the number on top of the pile is larger than the largest value seen so far, now 41. This time the value on top of the pile, 12, is not larger, so we do not want to save it. We simply throw it away and move on to the next number in the pile.

This compare-and-save-or-discard process continues until our original pile of numbers is empty, at which time the largest so far is the largest value in the entire list.

Let's see how we can convert this informal, pictorial solution into a formal algorithm that is built from the primitive operations shown in Figure 2.9.

We certainly cannot begin to search a list for a largest value until we have a list to search. Therefore, our first operation must be to get a value for *n*, the size of the list, followed by values for the *n*-element list , , …, . This can be done using our input primitive:
Get a value for *n*, the size of the list

Get values for , , . . ., , the list to be searched
Now that we have the data, we can begin to implement a solution.

Our informal description of the algorithm stated that we should begin by calling the first item in the list, , the largest value so far. (We know that this operation is meaningful because we stated that the list must always have at least one element.) We can express this formally as
Set the value of *largest so far* to
Our solution must also determine where that largest value occurs. To remember this value, let's create a variable called *location* to keep track of the position in the list where the largest value occurs. Because we have initialized *largest so far* to the first element in the list, we should initialize *location* to 1.

Set the value of *location* to 1

We are now ready to begin looking through the remaining items in list A to find the largest one. However, if we write something like the following instruction:

If the second item in the list is greater than *largest so far* then . . .

we will have made exactly the same mistake that occurred in the initial version of the sequential search algorithm shown in Figure 2.11. This instruction explicitly checks only the second item of the list. We would need to rewrite that statement to check the third item, the fourth item, and so on. Again, we are failing to use the idea of *iteration*, where we repetitively execute a loop as many times as it takes to produce the desired result.

To solve this problem, let's use the same technique used in the sequential search algorithm. Let's not talk about the second, third, fourth, . . . item in the list but about the *i*th item in the list, where *i* is a variable that takes on different values during the execution of the algorithm. Using this idea, a statement such as

If > *largest so far* then . . .
can be executed with different values for *i*. This allows us to check all *n* values in the list with a single statement. Initially, *i* should be given the value 2 because the first item in the list was automatically set to the largest value. Therefore, we want to begin our search with the second item in the list.

What operations should appear after the word *then*? A check of our earlier discussion shows that the algorithm must reset the values of both *largest so far* and *location*.

If is not larger than *largest so far*, then we do not want the algorithm to do anything. To indicate this, the if/then instruction can include an else clause that looks something like

This is certainly correct, but instructions that tell us not to do anything are usually omitted from an algorithm because they do not carry any meaningful information.

Regardless of whether the algorithm resets the values of *largest so far* and *location*, it needs to move on to the next item in the list. Our algorithm refers to , the *i*th item in the list, so it can move to the next item by simply adding 1 to the value of *i* and repeating the if/then statement. The outline of this iteration can be sketched as follows:

However, we do not want the loop to repeat forever. (Remember that one of the properties of an algorithm is that it must eventually halt.) What stops this iterative process? When does the algorithm display an answer and terminate execution?

The conditional operation "If > *largest so far* then . . ." is meaningful only if represents an actual element of list *A*. Because *A* contains n elements numbered 1 to *n*, the value of *i*must be in the range 1 to *n*. If $i > n$, then the loop has searched the entire list, and it is finished. Therefore, our continuation condition should be expressed as ($i \leq n$). When this condition becomes false, the algorithm can stop looping and print the values of both *largest so far* and *location*. Using our looping primitive, we can describe this iteration as follows:

We have now developed all the pieces of the algorithm and can finally put them together. Figure 2.14 shows the completed Find Largest algorithm. Note that the steps are not numbered. This omission is quite common, especially as algorithms get larger and more complex.

Figure 2.14Algorithm to find the largest value in a list

Practice Problems

1. What part(s) of the sequential search algorithm of Figure 2.13 would need to be changed if our phone book contained 1 million numbers rather than 10,000?

   Answer

   We would need to change Line 1 so that the algorithm would input 1 million numbers and names rather than 10,000. We would need to change Line 3 so that the loop would repeat a maximum of 1 million times rather than a maximum of 10,000. Actually, that is not a lot of change given that the problem is 100 times bigger. We only needed to change two lines. In fact, if the phone book were 1 billion numbers in length, then we would only need to change the same two lines.

2. Rewrite the sequential search algorithm to use the do/while looping structure shown in Figure 2.9 in place of the while structure.

   Answer

3. Modify the algorithm of Figure 2.14 so that it finds the smallest value in a list rather than the largest. Describe exactly what changes were necessary.

   Answer

   You must change the operation on Line 7 from a greater-than (>) to a less-than (<) sign. That line will now read as follows:

   That is the only required change. However, to avoid confusion about what the algorithm is doing, you probably should also change the name of the variable *largest so far* to something like *smallest so far* on

Lines 3, 7, 8, and 12. Otherwise, a casual reading of the algorithm might lead someone to think incorrectly that it is still an algorithm to find the largest value rather than the smallest.

4. Describe exactly what would happen to the algorithm in Figure 2.14 if you tried to apply it to an empty list of length . Describe exactly how you could fix this problem.
Answer

If (the list is empty), then there are no values for , , ..., . In particular, setting the value of *largest so far* to gives a meaningless value to *largest so far*. The while loop will not execute at all because *i* has the value 2 and *n* has the value 0, so the condition $i \leq n$ is false. The algorithm will print out nonsense values for *largest so far* and *location*.
The algorithm can be fixed by putting a conditional statement after Line 1. If , then the algorithm should print a message that says the list is empty. The "else" case will be the rest of the current algorithm.

5. Describe exactly what happens to the algorithm in Figure 2.14 when it is presented with a list with exactly one item, that is, . Determine whether this algorithm will or will not work correctly on this one-item list.
Answer

If , then *largest so far* is set to , the only list element, and *location* is set to 1. Also *i* is set to 2, so the while loop will not execute because it is false that $i \leq n$ (i.e., it is false that $2 \leq 1$). The correct values of *largest so far* and *location* are printed.

6. Would the Find Largest algorithm of Figure 2.14 still work correctly if the test on Line 7 were written as (≥ *largest so far*)? Explain why or why not.
Answer

Yes, it would still work correctly in one sense—it would find the largest numerical value in the list. However, if two numbers were equal, this change would cause the algorithm to find the last occurrence of that number rather than the first. So the value of *largest so far* would not change but the value of *location* might be different.

2.3.4 Example 4: Meeting Your Match

The last algorithm we develop in this chapter solves a common problem in computer science called *pattern matching*. For example, imagine that you have a large collection of Civil War data files that you want to use as resource material for an article on Abraham Lincoln. Your first step would probably be to search these files to locate every occurrence of the text patterns "Abraham Lincoln," "A. Lincoln," and "Lincoln." The process of searching for a special pattern of symbols within a larger collection of information is called *pattern matching*. Most good word processors provide this service

as a menu item called *Find* or something similar. Furthermore, most web search engines try to match your search terms to the keywords that appear on a webpage.

Pattern matching can be applied to almost any kind of information, including graphics, sound, and photographs. For example, an important medical application of pattern matching is to input an X-ray or CT scan image into a computer and then have the computer search for special patterns, such as dark spots, which represent conditions that should be brought to the attention of a physician. This can help speed up the interpretation of X-rays and avoid the problem of human error caused by fatigue or oversight. (Computers do not get tired or bored!)

One of the most interesting and exciting applications of pattern matching is to assist microbiologists and geneticists studying and analyzing the *human genome*, the basis for all human life. The Human Genome Project was started in 1990 with the goal of determining the sequence of chemical base pairs that comprise the human genome. The mammoth project was completed in April 2003, and today this genetic information is available via online databases to biological and medical researchers worldwide.

The human genome is composed of approximately 3.5 billion *nucleotides*, each of which can be one of only four different chemical compounds. These compounds (adenine, cytosine, thymine, and guanine) are usually referred to by the first letter of their chemical names: A, C, T, and G. Thus, the basis for our existence can be conceptually viewed as a very large "text file" written in a four-letter alphabet.

(If you were to write out the entire sequence of nucleotides, at 12 characters per inch, it would stretch from New York City to Moscow.) Sequences of these nucleotides are called *genes*. There are about 25,000 genes in the human genome, and they determine virtually all of our physical characteristics—sex, race, eye color, hair color, and height, to name just a few. Genes are also an important factor in the occurrence of certain diseases. A missing or flawed nucleotide can result in one of a number of serious genetic disorders, such as Down syndrome or Tay–Sachs disease. To help find a cure for these diseases, researchers attempt to locate individual genes that, when exhibiting a certain defect, cause a specific malady. The process of locating the starting and ending boundaries of an individual gene within the 3+ billion nucleotides is called *genome annotation*, and because of the scale of the problem it is done using automated algorithms that search DNA sequences in genomic databases—not unlike the algorithm we will develop in this section.

A gene is typically composed of thousands of nucleotides, and researchers generally do not know the entire sequence. However, they may know what a small portion of the gene—say, a few hundred nucleotides—looks like. Therefore, to search for one particular gene, they must match the sequence of nucleotides that they do know, called a *probe*, against the entire 3.5 billion-element genome to locate every occurrence of that probe. From this matching information, researchers hope to locate and isolate specific genes. For example,

When a match is found, researchers examine the nucleotides located before and after the probe to see whether they have located the desired gene and, if so, whether the gene is defective. Physicians hope someday to be able to "clip out" a bad sequence and insert in its place a correct sequence. (This goal took a giant step forward in 2016 with the development of CRISPR, a powerful genome-editing tool that makes cutting-and-pasting of DNA sequences far more precise and efficient.)

This application of pattern matching dispels any notion that the algorithms discussed here—sequential search (Figure 2.13), Find Largest (Figure 2.14), and pattern matching—are nothing more

than academic exercises that serve as examples for introductory classes but have absolutely no role in solving real-world problems. The algorithms that we have presented (or will present) *are* important, either in their own right or as building blocks for algorithms used by physical scientists, mathematicians, engineers, biologists, and social scientists.

Let's formally define the pattern-matching problem as follows:

You will be given some *text* composed of n characters that will be referred to as . (*Note*: n may be very, very large.) You will also be given a *pattern* of m characters, $m \leq n$, that will be represented as . (*Note*: m will usually be much, much smaller than n.) The algorithm must locate every occurrence of the given pattern within the text. The output of the algorithm is the location in the text where each match occurred. For this problem, the location of a match is defined to be the index position in the text where the match begins.
For example, if our text is the phrase "to be or not to be, that is the question" and the pattern for which we are searching is the word *to*, then our algorithm produces the following output:

The pattern-matching algorithm that we will implement is composed of two parts. In the first part, the pattern is aligned under a specific position of the text, and the algorithm determines whether there is a match at that given position. The second part of the algorithm "slides" the entire pattern ahead one character position. Assuming that we have not gone beyond the end of the text, the algorithm returns to the first part to check for a match at this new position. Pictorially, this algorithm can be represented as follows:

Repeat the following two steps.

The algorithm involves repetition of these two steps beginning at position 1 of the text and continuing until the pattern has slid off the right-hand end of the text.

A first draft of an algorithm that implements these ideas is shown in Figure 2.15, in which not all of the operations are expressed in terms of the basic algorithmic primitives of Figure 2.9. Although statements like "Set k, the starting location for the attempted match, to 1" and "Print the value of k, the starting location of the match" are just fine, the instructions "Attempt to match every character in the pattern beginning at position k of the text" and "Keep going until we have fallen off the end of the text" are certainly not primitives. On the contrary, they are high-level operations that, if written out using only the operations in Figure 2.9, would expand into many instructions.

Figure 2.15 First draft of the pattern-matching algorithm

Is it okay to use high-level statements like this in our algorithm? Wouldn't their use violate the requirement stated in Chapter 1 that algorithms be constructed only from unambiguous operations that can be directly executed by our computing agent?

In fact, it is perfectly acceptable, and quite useful, to use high-level statements like this during the *initial phase* of the algorithm design process. When starting to design an algorithm, we might not want to think only in terms of elementary operations such as input, computation, output, conditional, and iteration. Instead, we might want to express our proposed solution in terms of high-level and

broadly defined operations that represent dozens or even hundreds of primitive instructions. Here are some examples of these higher-level constructs:

Sort the entire list into ascending order.
Attempt to match the entire pattern against the text.
Find a root of the equation.
Using instructions like these in an algorithm allows us to postpone worrying about how to implement that operation and lets us focus instead on other aspects of the problem. This is equivalent to inserting the phrase "put a brief historical introduction right here" into a story that you are writing. It functions as a reminder of something that you must add but that can be postponed until a later time. With our algorithm, we will come back to these high-level constructs and either express them in terms of our available primitives or use existing building-block algorithms taken from a program library. However, we can do this at our convenience.

The use of high-level instructions during the design process is an example of one of the most important intellectual tools in computer science—**abstraction**. Abstraction refers to the separation of the high-level view of an entity or an operation from the low-level details of its implementation. It is abstraction that allows us to understand and intellectually manage any large, complex system, whether it is a mammoth corporation, a complex piece of machinery, or an intricate and very detailed algorithm. For example, the president of General Motors views the company in terms of its major corporate divisions and high-level policy issues, not in terms of every worker, every supplier, every car, or every bolt. Attempting to manage the company at that level of detail would drown the president in a sea of detail.

In computer science, we frequently use abstraction because of the complexity of hardware and software. For example, abstraction allows us to view the hardware component called "memory" as a single, indivisible high-level entity without paying heed to the billions of electronic devices that go into constructing a memory unit. (Chapter 4 examines how computer memories are built, and it makes extensive use of abstraction.) In algorithm design and software development, we use abstraction whenever we think of an operation at a high level and temporarily ignore how we might actually implement that operation. This allows us to decide which details to address now and which to postpone until later. Viewing an operation at a high level of abstraction and fleshing out the details of its implementation at a later time constitute an important computer science problem-solving strategy called **top-down design**.

Ultimately, however, we do have to describe how each of these high-level abstractions can be represented using the available algorithmic primitives. The fifth line of the first draft of the pattern-matching algorithm shown in Figure 2.15 reads:

Attempt to match every character in the pattern beginning at position *k* of the text

When this statement is reached, the pattern is aligned under the text beginning with the *k*th character. Pictorially, we are in the following situation:

The algorithm must now perform the following comparisons:

Compare to
Compare to
Compare to
.

.

.

Compare to
If every single one of these pairs is equal, then there is a match starting at position *k*. However, if even one pair is not equal, then there is no match, and the algorithm can immediately cease making comparisons at this location. Thus, we must construct a loop that executes until one of two things happens—it has either completed *m* successful comparisons (i.e., we have matched the entire pattern) or detected a mismatch. When either of these conditions occurs, the loop stops; however, if neither condition has occurred, the loop must keep going. Algorithmically, this iteration can be expressed in the following way. (Remember that *k* is the starting location in the text.)

When the loop has finished, we can determine whether there was a match by examining the current value of the variable *Mismatch*. If *Mismatch* is YES, then there was not a match because at least one of the characters was out of place. If *Mismatch* is NO, then every character in the pattern matched its corresponding character in the text, and there is a match starting at position *k*.

Regardless of whether there was a match at position *k*, we now must add 1 to *k* to begin searching for a match at the next position. This is the “sliding forward” step diagrammed earlier.

The final high-level statement in Figure 2.15 that needs to be expanded is the loop on Line 4.

Keep going until we have fallen off the end of the text

What does it mean to “fall off the end of the text”? Where is the last possible place that a match can occur? To answer these questions, let’s draw a diagram in which the last character of the pattern, , lines up directly under , the last character of the text.

This diagram illustrates that the last possible place a match could occur is when the first character of the pattern is aligned under the character at position of the text because is aligned under , is under , is aligned under , and so on. Thus, , which can be written as , is aligned under , which is . If we tried to slide the pattern forward any further, we would truly “fall off” the right-hand end of the text. Therefore, our loop must terminate when *k*, the starting point for the match, strictly exceeds the value of $n - m + 1$. We can express this as follows:
While ($k \leq (n - m + 1)$) do

Now we have all the pieces of our algorithm in place. We have expressed every statement in Figure 2.15 in terms of our basic algorithmic primitives and are ready to put it all together. The final draft of the pattern-matching algorithm is shown in Figure 2.16.

Figure 2.16Final draft of the pattern-matching algorithm

Hidden Figures

When it comes to telling the history of computer science, Hollywood often focuses on the contributions of white men to the exclusion of both women and people of color—Alan Turing (*The*

Imitation Game, 2014), Bill Gates (*Pirates of Silicon Valley*, 1999), Steven Jobs (*Jobs,* 2013; *Steve Jobs*, 2015), Mark Zuckerberg (*The Social Network*, 2010). That all changed in January, 2017 with the release of the movie *Hidden Figures*, the true story of three African-American women who worked as mathematicians and computer specialists at the NASA Space Research Center in Langley, Virginia in the early 1960s. (The movie received three Oscar nominations, including one for Best Picture.) These women, Katherine Johnson, Dorothy Vaughn, and Mary Jackson, were charged with carrying out the complex orbital calculations needed to send John Glenn into orbit and bring him back safely. Katherine Johnson went on to work on the launch of Apollo 11 to the moon and Apollo 13. She received the Presidential Medal of Freedom in 2015, and in 2016 the Langley Space Research Center was renamed the Katherine G. Johnson Computational Research Facility.

2.4 Conclusion

You have now had a chance to see the step-by-step design and development of some interesting, nontrivial algorithms. You have also been introduced to a number of fundamental concepts related to problem solving, including algorithm design and discovery, pseudocode, control statements, iteration, libraries, abstraction, and top-down design. However, this by no means marks the end of our discussion of algorithms. The development of a correct solution to a problem is only the first step in creating a useful piece of software.

Designing a technically correct algorithm to solve a given problem is only part of what computer scientists do. They must also ensure that they have created an *efficient* algorithm that generates results quickly enough for its intended users. Chapter 1 described a brute force chess algorithm that would, at least theoretically, play perfect chess but that would be unusable because it would take millions of centuries to make its first move. Similarly, a reverse directory lookup program that takes 30 minutes to locate a person's name would be of little or no use. The caller would surely have hung up long before we learned who it was. This practical concern for efficiency and usefulness, in addition to correctness, is one of the hallmarks of computer science.

Therefore, after developing a correct algorithm, we must analyze it thoroughly and study its efficiency properties and operating characteristics. We must ask ourselves how quickly it will give us the desired results and whether it is better than other algorithms that solve the same problem. This analysis, which is the central topic of Chapter 3, enables us to create algorithms that are not only correct but elegant, efficient, and useful as well.

Practice Problems

1. Consider the following "reverse telephone directory."

Number	***Name***
(648) 555-1285	Smith

(247) 834-6543 Jones
(771) 921-5281 Adams
(356) 327-8900 Doe

2. Trace the sequential search algorithm of Figure 2.13 using each of the following *NUMBERS*and show the output produced.

1. (771) 921-5281

   Answer

2. (488) 351-1673

   Answer

3. Consider the following list of seven data values.

   22, 18, 23, 17, 25, 30, 2

   Trace the Find Largest algorithm of Figure 2.14 and show the output produced.

   Answer
   , , , , , , ,
   Output: . .

4. Consider the following text.

   Text: A man and a woman

   Trace the pattern-matching algorithm of Figure 2.16 using the two-character pattern *an* and show the output produced.

   Answer
   . The pattern has 2 characters.
   . The text has 17 characters.
   Output: There is a match at position 4

   In a similar way, the program will produce the following two additional lines of output:

   There is a match at position 7

   There is a match at position 16

5. Explain exactly what would happen to the algorithm of Figure 2.16 if *m*, the length of the pattern, were greater than *n*, the length of the text.

   Answer
   If $m > n$, then $n - m + 1 \leq 0$. Because the value of k is set to 1 right before the outer while loop, the condition $k \leq (n - m + 1)$ is false, the loop is not executed, and the algorithm terminates with no output.

6. Explain exactly what would happen to the algorithm of Figure 2.16 if Line 5 had been incorrectly written as “set the value of *i* to 0”.

Answer

If we had incorrectly initialized *i* to 0 rather than 1, then the algorithm would halt with a fatal error on Line 8. In that line, we reference the variable . If *i* is 0, this would be , but there is no such value, as the pattern begins with the element . This shows how important it is to be very careful about not only the "big issues" but the little details as well. Even a very tiny initialization error can cause an entire algorithm to fail.

7. Determine whether the pattern-matching algorithm of Figure 2.16 will work if *n*, the length of the text, is exactly the same size as *m*, the length of the pattern.

Answer

Yes, it will work correctly. The quantity $(n - m + 1)$ evaluates to 1. Since k is initialized to 1, the test $k \leq (n - m + 1)$ will initially evaluate to True and the loop will be executed. The text and the pattern will be matched beginning at position 1. Then when *k* is incremented to 2, the test will become false and we will exit the loop, as desired.