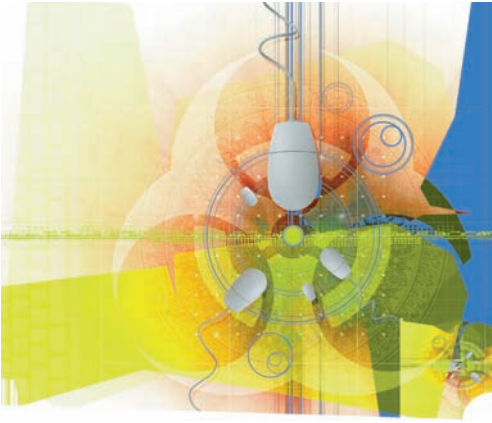
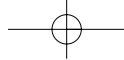


CHAPTER 2

Algorithm Discovery and Design

- 2.1 Introduction
 - 2.2 Representing Algorithms
 - 2.2.1 Pseudocode
 - 2.2.2 Sequential Operations
 - 2.2.3 Conditional and Iterative Operations
 - 2.3 Examples of Algorithmic Problem Solving
 - 2.3.1 Example 1: Go Forth and Multiply
 - 2.3.2 Example 2: Looking, Looking, Looking
 - LABORATORY EXPERIENCE 2**
 - 2.3.3 Example 3: Big, Bigger, Biggest
 - LABORATORY EXPERIENCE 3**
 - 2.3.4 Example 4: Meeting Your Match
 - 2.4 Conclusion
- EXERCISES**
- CHALLENGE WORK**
- FOR FURTHER READING**



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 VCR (Figure 1.1) and washing your hair (Figure 1.3). While these are perfectly valid examples of algorithms, they are not of much interest to computer scientists. This chapter develops more fully the notions of algorithm 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.

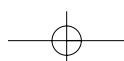
2.2 Representing Algorithms

▶ 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. (In our case it is English, but it could be 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 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.

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 may be wonderful for 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 inside the text. For example, on the eighth line of 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 on the second line that starts, “When these initializations have



**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 a_i and b_i and the current value of the carry digit to get the result called c_i . Now check the value of c_i to see whether it is greater than or equal to 10. If c_i is greater than or equal to 10, then reset the value of *carry* to 1 and reduce the value of c_i by 10; otherwise, set the value of *carry* to zero. 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 c_m to the value of *carry* and print out the final result, which consists of the digits $c_m c_{m-1} \dots c_0$. After printing the result, the algorithm is finished, and it terminates.

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 is disastrous when writing algorithms that must always execute in the same way and produce identical results. We can see an example of this problem in the sentence on lines 7 and 8 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 may 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 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.

As an algorithmic design language, this notation is also seriously flawed. During the initial phases of design, we should be thinking and writing at a highly abstract level. Using a programming language to express our design forces us to deal immediately with 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 cryptic statement “int[] a = new int[100];”? These technical details clutter our thoughts, and at

**FIGURE 2.2**

The Beginning of the Addition Algorithm of Figure 1.2 Expressed in a High-Level Programming Language

```
{
int i, m, carry;
int[] a = new int[100];
int[] b = new int[100];
int[] c = new int[100];
m = Console.readInt();
for (int j = 0; j <= m-1; j++) {
    a[j] = Console.readInt();
    b[j] = Console.readInt();
}
carry = 0;
i = 0;
while (i < m) {
    c[i] = a[i] + b[i] + carry;
    if (c[i] >= 10)
        .
        .
        .
}
```

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 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 called a programming language without any 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, and 1.3(a) and (b) are all written in pseudocode.

In the following sections we will introduce a set of popular and easy-to-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 to a mailbox into which one can place a value and from which one 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, it is discarded and replaced by the new value 0. You can visualize the result of this operation as follows:

carry 0

Here is another example:

Set the value of *Area* to (πr^2)

Assuming that the variable *r* has been given a value by a previous instruction in the algorithm, this statement evaluates the arithmetic expression πr^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:

- Step 4:** Add the two digits a_i and b_i to the current value of *carry* to get c_i
- Step 6:** Add 1 to *i*, effectively moving one column to the left
- 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:

- Step 4:** Set the value of c_i to $(a_i + b_i + \textit{carry})$
- Step 6:** Set the value of *i* to $(i + 1)$
- 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 really care about the minor language differences between

Add a and b to get c

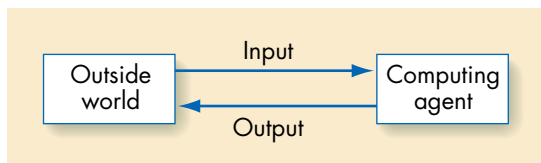
and

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 can assume that the computing agent executing your algorithm has all the capabilities of a typical calculator. Therefore, it “knows” how to do all basic arithmetic operations such as $+$, $-$, \times , \div , square root, 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 submit to the computing agent 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 system (e.g., keyboard, screen, mouse, printer, hard drive, CD). However, when designing algorithms, we generally do not concern ourselves with such details. We care only that data is provided 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 on a screen 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 quotes’ to enclose messages so as to distinguish them from such pseudocode constructs as “variable” and “arithmetic expression,” which are enclosed in double quotes.)

Print the message ‘Sorry, no answers were computed.’

Using the 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.

PRACTICE PROBLEMS

Write pseudocode versions of

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



FIGURE 2.3

Algorithm for Computing Average Miles per Gallon

Average Miles per Gallon Algorithm (Version 1)

STEP	OPERATION
1	Get values for <i>gallons used</i> , <i>starting mileage</i> , <i>ending mileage</i>
2	Set value of <i>distance driven</i> to (<i>ending mileage</i> – <i>starting mileage</i>)
3	Set value of <i>average miles per gallon</i> to (<i>distance driven</i> ÷ <i>gallons used</i>)
4	Print the value of <i>average miles per gallon</i>
5	Stop

▶ 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, most real-world problems are not straight-line. 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.

Conditional statements are the “question-asking” operations of an algorithm. They allow an algorithm to ask a question and to 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*, which has the following format:

```

If “a true/false condition” is true then
    first set of algorithmic operations
Else (or otherwise)
    second set of algorithmic operations
  
```

The meaning of this statement is as follows:

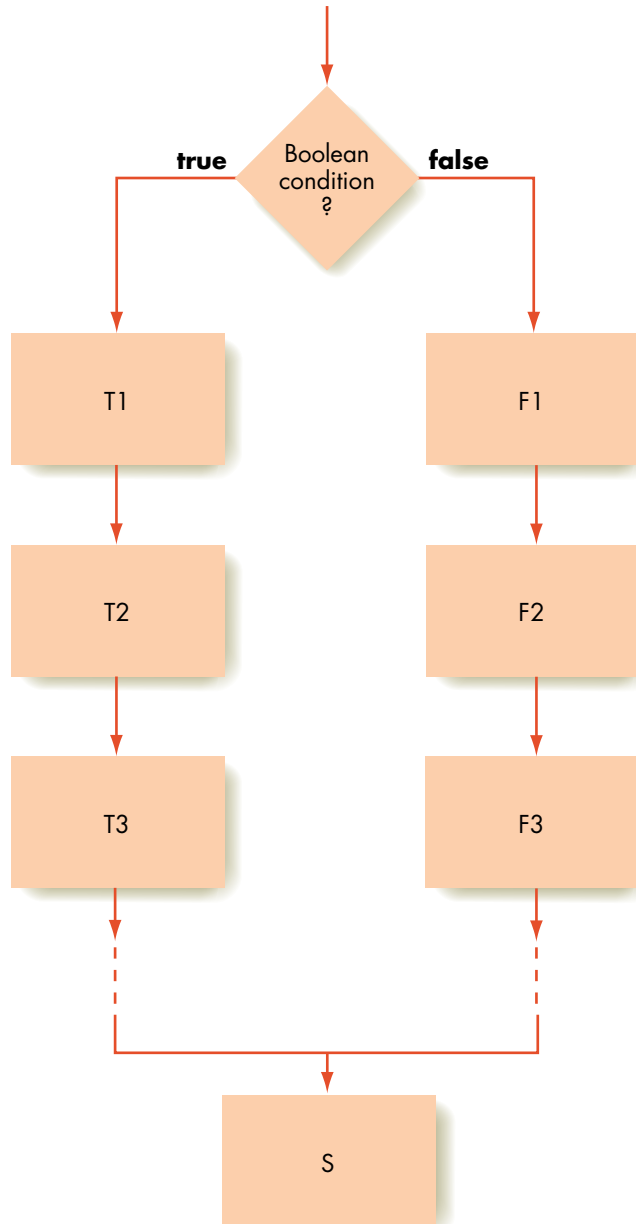
1. Evaluate the true/false condition on the first line to determine whether it is true or false.
2. If the condition is true, then do the first set of algorithmic operations and skip the second set entirely.
3. If the condition is false, then skip the first set of operations and do the second set.
4. 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*.

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 of Figure 1.2. (The statement has been reformatted slightly to highlight the two alternatives clearly, but it has not been changed.)

**FIGURE 2.4**

The *If/Then/Else* Pseudocode Statement



If $(c_i \geq 10)$ then
 Set the value of c_i to $(c_i - 10)$
 Set the value of *carry* to 1
 Else
 Set the value of *carry* to 0

The condition $(c_i \geq 10)$ 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 greater than 25.0 mpg.

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 1,000,000 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 can carry out that same task in less than 1 second!

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

```
While ("a true/false condition") do step i to step j
    step i:      operation
    step i + 1:  operation
    .
    .
    .
    step j:      operation
```

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



FIGURE 2.5

Second Version of the Average Miles per Gallon Algorithm

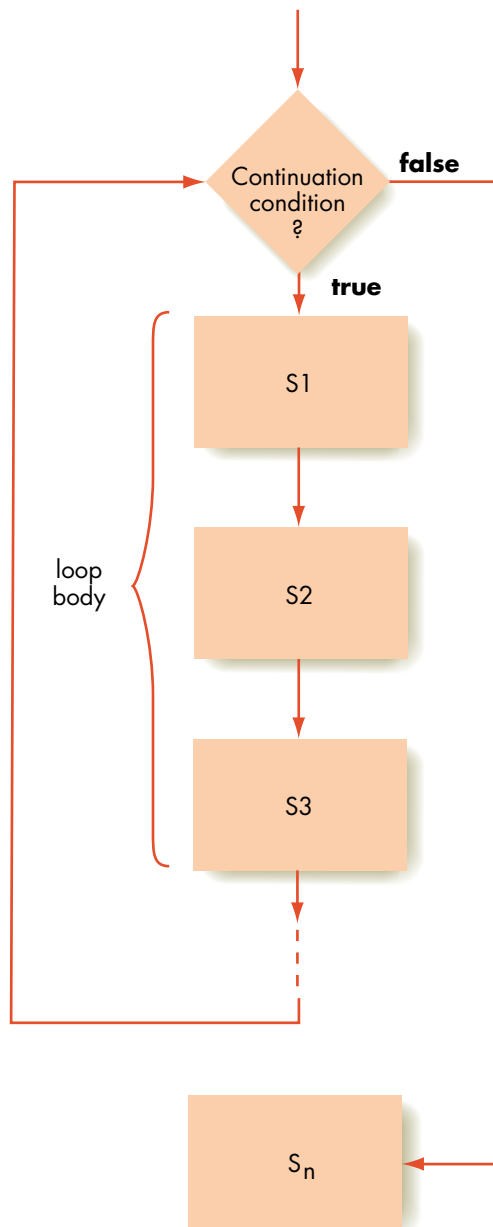
Average Miles per Gallon Algorithm (Version 2)

STEP	OPERATION
1	Get values for <i>gallons used</i> , <i>starting mileage</i> , <i>ending mileage</i>
2	Set value of <i>distance driven</i> to (<i>ending mileage</i> – <i>starting mileage</i>)
3	Set value of <i>average miles per gallon</i> to (<i>distance driven</i> ÷ <i>gallons used</i>)
4	Print the value of <i>average miles per gallon</i>
5	If <i>average miles per gallon</i> is greater than 25.0 then
6	Print the message ‘You are getting good gas mileage’
	Else
7	Print the message ‘You are NOT getting good gas mileage’
8	Stop

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 previous diagram. 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 S_1 , S_2 , S_3 , . . . , 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 2.6
Execution of the While Loop



Here is a simple example of a loop:

Step	Operation
1	Set the value of <i>count</i> to 1
2	While (<i>count</i> ≤ 100) do step 3 to step 5
3	Set <i>square</i> to (<i>count</i> x <i>count</i>)
4	Print the values of <i>count</i> and <i>square</i>
5	Add 1 to <i>count</i>

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, 4, and 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 is 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(a). 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 algorithm until the response provided by the user is No. (Note that the algorithm must initialize the value of response to Yes, since the very first thing that the loop does is test the value of this quantity.)

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 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:



FIGURE 2.7

Third Version of the Average
Miles per Gallon Algorithm

Average Miles per Gallon Algorithm (Version 3)

STEP	OPERATION
1	<i>response</i> = Yes
2	While (<i>response</i> = Yes) do steps 3 through 11
3	Get values for <i>gallons used</i> , <i>starting mileage</i> , <i>ending mileage</i>
4	Set value of <i>distance driven</i> to (<i>ending mileage</i> – <i>starting mileage</i>)
5	Set value of <i>average miles per gallon</i> to (<i>distance driven</i> ÷ <i>gallons used</i>)
6	Print the value of <i>average miles per gallon</i>
7	If average miles per gallon > 25.0 then
8	Print the message 'You are getting good gas mileage'
	Else
9	Print the message 'You are NOT getting good gas mileage'
10	Print the message 'Do you want to do this again? Enter Yes or No'
11	Get a new value for <i>response</i> from the user
12	Stop

```

While ("a true/false condition") do
    operation
    .
    .
    .
    operation
End of the loop

```

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".

This type of loop 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 we ask the user if they want to solve the problem again, but we ask that at the very end of execution of the loop body. Therefore, we had to give the variable called *response* a "dummy" value of Yes so that the test would be meaningful when the loop was first entered.

Another variation of the looping structure is the **posttest loop**, which also uses a true/false continuation condition to control execution of the loop. However, 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:

```

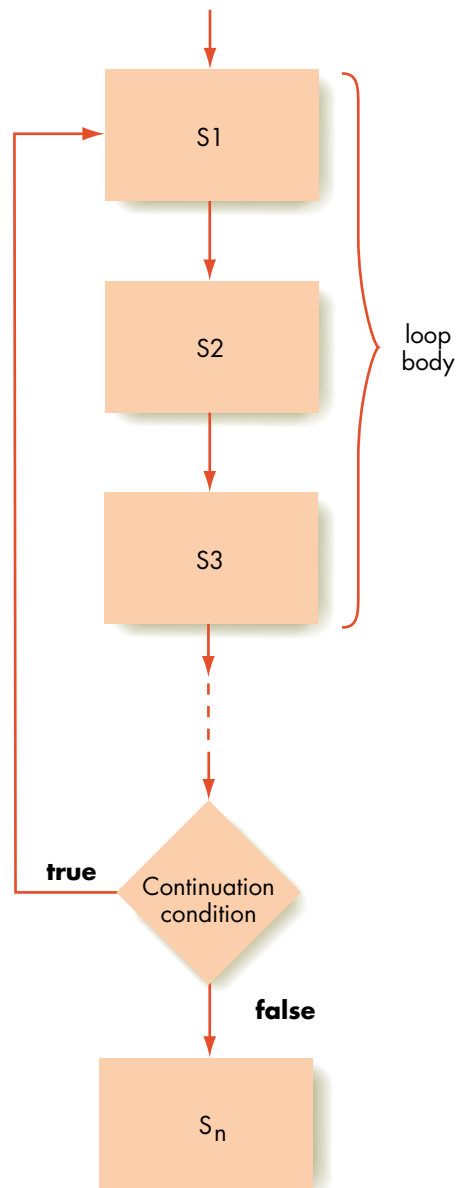
Do
    operation
    operation
    .
    .
    .
While ("a true/false condition")

```

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 it 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, while the While loop can execute 0, 1, or more times. Figure 2.8 diagrams the execution of the posttest Do/While looping structure.

Figure 2.9 summarizes the algorithmic operations introduced in this section. These operations represent the **primitives** 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 2.8
*Execution of the Do/While
Posttest Loop*



**FIGURE 2.9**

*Summary of Pseudocode
Language Instructions*

COMPUTATION:

Set the value of “variable” to “arithmetic expression”

INPUT/OUTPUT:

Get a value for “variable”, “variable”. . .

Print the value of “variable”, “variable”, . . .

Print the message ‘message’

CONDITIONAL:

If “a true/false condition” is true then

 first set of algorithmic operations

Else

 second set of algorithmic operations

ITERATIVE:

While (“a true/false condition”) do step *i* through step *j*

 Step *i*: operation

 .

 .

 .

 Step *j*: operation

While (“a true/false condition”) do

 operation

 .

 .

 .

 operation

End of the loop

Do

 operation

 operation

 .

 .

 .

While (“a true/false condition”)

From Little Primitives Mighty Algorithms Do Grow

Although the set of algorithmic primitives shown in Figure 2.9 may 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 may 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 1.2 but also the

exceedingly complex algorithms needed to fly NASA’s space shuttles, run the international telephone switching system, and describe 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 a huge set of primitives. It can also arise from a small number of simple building blocks combined in interesting ways. This is the real secret of building algorithms.

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.)
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 zero or negative, your algorithm should not compute the average but should print the error message 'Bad Data' instead.
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.
4. Write an algorithm that gets as input a single data value x and outputs the three values x^2 , $\sin x$, and $1/x$. This process is repeated until the input value for x is equal to 999, at which time the algorithm terminates.
5. Write an algorithm that inputs the length and width, in feet, of a rectangular carpet and the price of the carpet in \$/square yard. It then determines if we can afford to purchase this carpet, given that our total budget for carpeting is \$500.

2.3 Examples of Algorithmic Problem Solving

▶ 2.3.1 Example 1: Go Forth and Multiply

Our first example of algorithmic problem solving addresses a problem originally posed in Chapter 1 (Exercise 9). That problem asked you to implement an algorithm to multiply two numbers using repeated addition. This problem can be formally expressed as follows:

Given 2 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 $a + a + a + \dots + a$ (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

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. Since we want to stop when we have completed the loop b times, we want to stop when ($count = b$). 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 $(b - 1)$.

```

Get values for  $a$  and  $b$ 
Set the value of  $count$  to 0
While ( $count < b$ ) do
    ... the rest of the loop body will go here ...
    Set the value of  $count$  to ( $count + 1$ )
End of loop

```

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* added to 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:

```

Get values for  $a$  and  $b$ 
Set the value of  $count$  to 0
Set the value of  $product$  to 0
While ( $count < b$ ) do
    Set the value of  $product$  to ( $product + a$ )
    Set the value of  $count$  to ( $count+1$ )
End of loop

```

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 stored it 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: $a \geq 0$ and $b \geq 0$. The above algorithm works for positive values of *a* and *b*, but what happens when either $a = 0$ or $b = 0$? Does it still function correctly?

If $b = 0$ 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 printed, which is the correct answer.

Now let's look at what happens when $a = 0$ and *b* is any non-zero value, say 5,386. Of course we know immediately that the correct result is 0, but the 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*. Since adding 0 to anything has no effect, *product* remains at 0, and that is the output that is printed. 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 real-world 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 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 result to 0 without requiring any further computations:

```
If (either  $a = 0$  or  $b = 0$ ) then
    Set the value of product to 0
Else
    ... solve the problem as described above ...
```

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.

This first example needed only 2 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 lists of names, sequences of characters, or sets of experimental data. In the following sections we will show examples of the types of processing—searching, reordering, comparing—often done on these large collections of information.

**FIGURE 2.10**

Algorithm for Multiplication via Repeated Addition

Multiplication via Repeated Addition

```

Get values for a and b
If (either  $a = 0$  or  $b = 0$ ) then
    Set the value of product to 0
Else
    Set the value of count to 0
    Set the value of product to 0
    While ( $count < b$ ) do
        Set the value of product to ( $product + a$ )
        Set the value of count to ( $count+1$ )
    End of loop
Print the value of product
Stop
  
```

PRACTICE PROBLEMS

1. Manually work through the algorithm in Figure 2.10 using the input values $a = 2$, $b = 4$. After each completed pass through the loop, write down the current value of the four variables a , b , $count$, and $product$.
2. 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:
 case 1: $a = -2$, $b = 4$
 case 2: $a = 2$, $b = -4$
3. If the algorithm of Figure 2.10 produced the wrong answer for either case 1 or case 2 of question 2, explain exactly how you could fix the algorithm so it works correctly and produces the correct answer.

▶ 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 will solve was also mentioned in Chapter 1—locating a particular person’s name in a telephone book. This is just the type of simple and rather uninteresting repetitive task so well suited to computerization. (Many large telephone companies have implemented such an application. Most of us have dialed directory assistance and heard the desired telephone number spoken in a computer-generated voice.)

Assume that we have a list of 10,000 names that we define as $N_1, N_2, N_3, \dots, N_{10,000}$, along with the 10,000 telephone numbers of those individuals, denoted as $T_1, T_2, T_3, \dots, T_{10,000}$. To simplify the problem, we initially assume that all names in the book are unique and that the names need not be in alphabetical order. Essentially what we have described is a nonalphabetized telephone book of the following form:

<i>Name</i>	<i>Telephone Number</i>	
N_1	T_1	}
N_2	T_2	
N_3	T_3	
.	.	
.	.	
.	.	
.	.	
.	.	
$N_{10,000}$	$T_{10,000}$	

Let’s create an algorithm that allows us to input the name of a specific person, which we will denote as *NAME*. The algorithm will check to see if *NAME* matches any of the 10,000 names contained in our telephone book. If *NAME* matches the value N_j , where j is a value between 1 and 10,000, then the output of our algorithm will be the telephone number of that person: the value T_j . If *NAME* is not contained in our telephone book, then the output of our algorithm will be the message “I am sorry but this name is not in the 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 names in our telephone book are not in alphabetical order, there is no clever way to speed up the search. With a random collection of names, there is no method more efficient than starting at the beginning and looking at each name in the list, one at a time, until we either find the one we are looking for or 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 telephone book to see who lives at 123 Elm Street, because a telephone book is not sorted by address. It is also the way that we look through 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.

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 *NAME* in the very first position of the telephone book, N_1 , then we get the answer T_1 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 *NAME* in positions $N_2, \dots, N_{10,000}$. Of



FIGURE 2.11

First Attempt at Designing a Sequential Search Algorithm

STEP	OPERATION
1	Get values for $NAME$, N_1 , \dots , $N_{10,000}$, and $T_1, \dots, T_{10,000}$
2	If $NAME = N_1$ then print the value of T_1
3	If $NAME = N_2$ then print the value of T_2
4	If $NAME = N_3$ then print the value of T_3
.	.
.	.
.	.
10,000	If $NAME = N_{9,999}$ then print the value of $T_{9,999}$
10,001	If $NAME = N_{10,000}$ then print the value of $T_{10,000}$
10,002	Stop

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 $NAME$ is not in the list, this algorithm simply stops (at step 10,002) rather than providing the desired result, a message that the name you requested is not in the directory. 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 called **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(a) and (b). 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 names. That is, N_i refers to the i th name 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 names and pointing to the one on which the algorithm is currently working.



FIGURE 2.12

Second Attempt at Designing a Sequential Search Algorithm

STEP	OPERATION
1	Get values for $NAME$, $N_1, \dots, N_{10,000}$, and $T_1, \dots, T_{10,000}$
2	Set the value of i to 1 and set the value of $Found$ to NO
3	While ($Found = \text{NO}$) do steps 4 through 7
4	If $NAME$ is equal to the i th name on the list N_i then
5	Print the telephone number of that person, T_i
6	Set the value of $Found$ to YES
	Else ($NAME$ is not equal to N_i)
7	Add 1 to the value of i
8	Stop

The first time through the loop, the value of the index i is 1, so the algorithm checks to see whether $NAME$ is equal to N_1 , the first name on the list. If it is, then the algorithm writes out the result and sets $Found$ to YES, which causes the loop in steps 4 through 7 to terminate. If it is not the desired $NAME$, 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 (in step 4) to see whether $NAME$ is equal to N_2 , the second name on the list. In this way, the algorithm uses the single conditional statement “If $NAME$ is equal to the i th name on the list . . .” to check up to 10,000 names. 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 $NAME$ does not appear anywhere on the list. This final problem can be solved by terminating the loop when the desired name is found or the end of the list is reached. 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 $NAME$. 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 telephone book look up 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, rather than the 10,002 steps of the first attempt in Figure 2.11. As you can see, not all algorithms are created equal.

Looking at the algorithm in Figure 2.13, our first thought may be that this is not at all how people manually search a telephone book. When looking for a particular telephone number, we would never turn to page 1, column 1, and scan all names beginning with Aardvark, Alan. Certainly, a telephone company in New York City would not be satisfied with the performance of a



FIGURE 2.13

The Sequential Search Algorithm

Sequential Search Algorithm

STEP	OPERATION
1	Get values for $NAME$, $N_1, \dots, N_{10,000}$, and $T_1, \dots, T_{10,000}$
2	Set the value of i to 1 and set the value of $Found$ to NO
3	While both ($Found = \text{NO}$) and ($i \leq 10,000$) do steps 4 through 7
4	If $NAME$ is equal to the i th name on the list N_i , then
5	Print the telephone number of that person, T_i
6	Set the value of $Found$ to YES
	Else ($NAME$ is not equal to N_i)
7	Add 1 to the value of i
8	If ($Found = \text{NO}$) then
9	Print the message ‘Sorry, this name is not in the directory’
10	Stop

directory search algorithm that always began on page 1 of its 2,000-page telephone book.

Because our telephone book was not alphabetized, we really had no choice in the design of a search algorithm. However, in real life we can do much better than sequential search, because telephone books *are* alphabetized, and we can exploit this fact during the search process. For example, we know that *M* is about halfway through the alphabet, so when looking for the name Samuel Miller, we open the telephone book somewhere in the middle rather than to the first page. We then see exactly where we are by looking at the first letter of the names on the current page, and then we move forward or backward toward names beginning with *M*. This approach allows us to find the desired name much more quickly than searching sequentially beginning with the letter *A*.


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

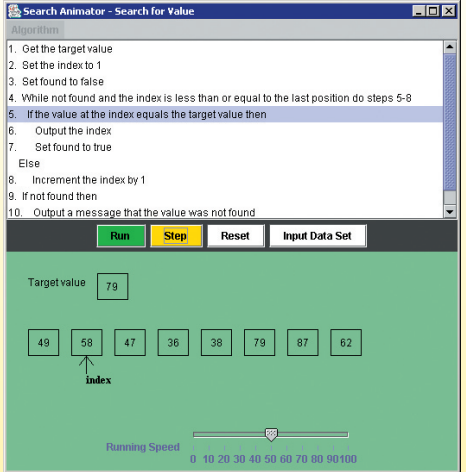
The selection of an algorithm to solve a problem is greatly influenced by the way the data for that problem are organized.

An algorithm is a method for processing some data to produce a result, and the way the data are 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 *alphabetized* telephone books that is far superior to the one shown in Figure 2.13.

LABORATORY EXPERIENCE 2





Computer science is an empirical discipline as well as a theoretical one. Learning comes not just from reading about concepts like algorithms, but manipulating and observing them as well. The laboratory manual for this text includes laboratory exercises that enable you to engage the ideas and concepts presented on these pages. Laboratory Experience 2 introduces the concept of *algorithm animation*, in which you can observe an algorithm being executed and watch as data values are dynamically transformed into final results. Here is an example of the type of output produced by this Laboratory Experience.

Bringing an algorithm to life in this way can help you understand what the algorithm does and how it works. The first animation that you will work with is the sequential search algorithm shown in Figure 2.13. The laboratory software allows you to create a list of data values, and to watch as the algorithm searches this list to determine whether a special target value occurs.

We strongly encourage you to work through these laboratory experiences to deepen your understanding of the ideas presented in this and following chapters.

▶ 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 in 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 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 may 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 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 A_1, A_2, \dots, A_n , find and print out both the largest value in the list and the position in the list where that largest value occurred.

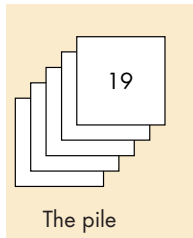
For example, if our list contained the five values

19, 41, 12, 63, 22 ($n = 5$)

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 solution as a formal algorithm.

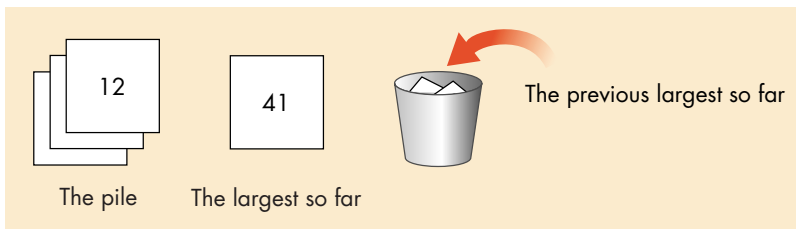
For example, suppose we were given a pile of papers, each of which contains a single number, and were asked to locate the largest number in the pile? (The following diagrams assume the papers contain the five values 19, 41, 12, 63, and 22.)



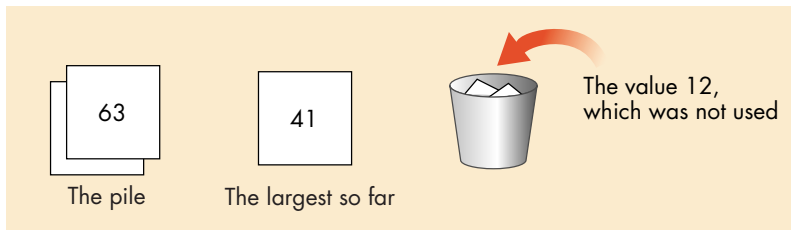
We might start off by saying that the first number in the pile (the top one) is the largest one that we have seen so far and then putting it off to the side where we are keeping the largest value.



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 off 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 A_1, A_2, \dots, A_n . This can be done using our input primitive:

Get a value for n , the size of the list

Get values for A_1, A_2, \dots, A_n , 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, A_1 , the largest value so far. (We know that this operation is meaningful since 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 A_1

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 $A_i > \textit{largest so far}$ 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.

.
.
.
Set the value of i to 2

.
.
.
If $A_i > \textit{largest so far}$ then . . .

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

If $A_i > \textit{largest so far}$ then
 Set *largest so far* to A_i
 Set *location* to i

If A_i 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

Else
 Don't do anything at all to *largest so far* and *location*

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.

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 A_i , 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:

→ If $A_i > \textit{largest so far}$ then
 Set *largest so far* to A_i
 Set *location* to i
 Add 1 to the value of i
.
.
.

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 $A_i > \textit{largest so far}$ then . . .” is meaningful only if A_i 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. The algorithm can now print the values of both *largest so far* and *location*. Using our looping primitive, we can describe this iteration as follows:

```
While ( $i \leq n$ ) do
  If  $A_i > \textit{largest so far}$  then
    Set largest so far to  $A_i$ 
    Set location to  $i$ 
  Add 1 to the value of  $i$ 
End of the loop
```

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 2.14

Algorithm to Find the Largest Value in a List

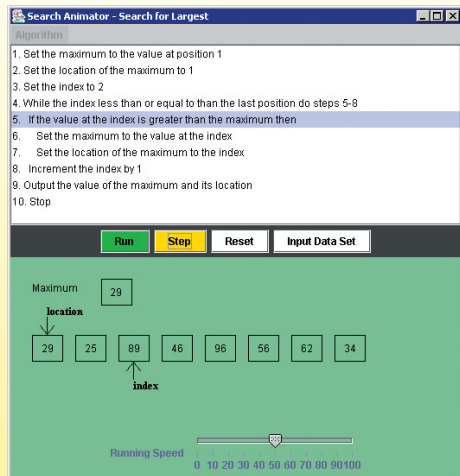
Find Largest Algorithm

```
Get a value for  $n$ , the size of the list
Get values for  $A_1, A_2, \dots, A_n$ , the list to be searched
Set the value of largest so far to  $A_1$ 
Set the value of location to 1
Set the value of  $i$  to 2
While ( $i \leq n$ ) do
  If  $A_i > \textit{largest so far}$  then
    Set largest so far to  $A_i$ 
    Set location to  $i$ 
  Add 1 to the value of  $i$ 
End of the loop
Print out the values of largest so far and location
Stop
```

PRACTICE PROBLEMS

1. 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.
2. Describe exactly what would happen to the algorithm in Figure 2.14 if you tried to apply it to an empty list of length $n = 0$. Describe exactly how you could fix this problem.
3. Describe exactly what happens to the algorithm in Figure 2.14 when it is presented with a list with exactly one item, i.e., $n = 1$.

LABORATORY EXPERIENCE 3



Like Laboratory Experience 2, this laboratory also uses animation to help you better understand the concept of algorithm design and execution. It presents an animation of the Find Largest algorithm discussed in the text and shown in Figure 2.14. An example of what you will see on the screen when you run this lab is shown here.

This laboratory experience allows you to create a list of data and watch as the algorithm attempts to determine the largest numerical value contained in that list. You will be able to observe dynamic changes to the variables index, location, and maximum, and will be able to see how values are set and discarded as the algorithm executes. Like the previous laboratory experience, it is intended to give you a deeper understanding of how this algorithm works by allowing you to observe its behavior.

▶ 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 collection of Civil War data files that you wish 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 keys to the keywords that appear on a Web page.

Pattern matching can be applied to almost any kind of information, including graphics, sound, and pictures. 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 assisting microbiologists and geneticists studying and mapping the *human genome*, the basis for all human life. The human genome is composed of a

sequence of approximately 3.5 billion *nucleotides*, each of which can be one of only four different chemical compounds. These compounds (adenine, cytosine, thymine, 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 rendered in a very large “text file” written in a four-letter alphabet.

... *T C G G A C T A A C A T C G G G A T C G A G A T G* ...

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 are attempting to map the entire human genome—to locate individual genes that, when exhibiting a certain defect, cause a specific malady. 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 be able to isolate specific genes. For example,

Genome: ... *T C A G G C T A A T C G T A G G* ...

Probe: *T A A T C* a match

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, to see 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 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, 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 $T_1 T_2 \dots T_n$. You will also be given a *pattern* of m characters, $m \leq n$, that will be represented as $P_1 P_2 \dots P_m$. The algorithm must locate every occurrence of the 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:

Text: to be or not to be, that is the question
 Pattern: to
 Output: Match starting at position 1.

Text: to be or not to be, that is the question
 Pattern: to
 Output: Match starting at position 14. (The t is in position 14,
 including blanks.)

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.

STEP 1: The matching process:

$$\begin{array}{cccccc} T_1 & T_2 & T_3 & T_4 & T_5 & \dots \\ P_1 & P_2 & P_3 & & & \end{array}$$

STEP 2: The slide forward:

$$\begin{array}{cccccc} T_1 & T_2 & T_3 & T_4 & T_5 & \dots \\ 1\text{-character slide} \rightarrow & & P_1 & P_2 & P_3 & \end{array}$$

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. While 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 both high-level operations that, if written out using only the operations in Figure 2.9, would expand into many instructions.

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 may not want to think only in terms of elementary operations such as input, computation, output, conditional, and iteration. Instead, we may want to express our proposed solution in terms of high-level



FIGURE 2.15

First Draft of the Pattern-Matching Algorithm

```

Get values for  $n$  and  $m$ , the size of the text and the pattern, respectively
Get values for both the text  $T_1 T_2 \dots T_n$  and the pattern  $P_1 P_2 \dots P_m$ 
Set  $k$ , the starting location for the attempted match, to 1
Keep going until we have fallen off the end of the text
    Attempt to match every character in the pattern beginning at
        position  $k$  of the text (this is step 1 from the previous page)
    If there was a match then
        Print the value of  $k$ , the starting location of the match
        Add 1 to  $k$ , which slides the pattern forward one position (this is step 2)
End of the loop
Stop

```

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. Eventually, we will come back to these statements 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 very high-level policy issues, not in terms of every worker, every supplier, and every car. 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. 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 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:

<i>Text:</i>	$T_1 T_2 T_3 \dots$	$T_k T_{k+1} T_{k+2} \dots T_{k+(m-1)} \dots$
<i>Pattern:</i>		$P_1 P_2 P_3 \dots P_m$

The algorithm must now perform the following comparisons:

Compare P_1 to T_k
 Compare P_2 to T_{k+1}
 Compare P_3 to T_{k+2}
 .
 .
 .
 Compare P_m to $T_{k+(m-1)}$

If the members of every single one of these pairs are equal, then there is a match. 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 it has detected a mismatch. When either of these conditions occurs the algorithm stops; however, if neither condition has occurred, the algorithm must keep going. Algorithmically, this iteration can be expressed in the following way. (Remember that k is the starting location in the text.)

```

Set the value of  $i$  to 1
Set the value of  $Mismatch$  to NO
While both  $(i \leq m)$  and  $(Mismatch = NO)$ 
  If  $P_i \neq T_{k+(i-1)}$  then
    Set  $Mismatch$  to YES
  Else
    Increment  $i$  by 1 (to move to the next character)
End of the loop
  
```

When the loop has finished, we can determine whether there has been 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.

```

If  $Mismatch = NO$  then
  Print the message 'There is a match at position'
  Print the value of  $k$ 
  
```

Regardless of whether there was a match at position k , we 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, P_m , lines up directly under T_n , the last character of the text.

<i>Text:</i>	T_1	T_2	T_3	\dots	T_{n-m+1}	\dots	T_{n-2}	T_{n-1}	T_n
<i>Pattern:</i>					P_1	\dots	P_{m-2}	P_{m-1}	P_m

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 T_{n-m+1} of the text, because P_m is aligned under T_n , P_{m-1} is under T_{n-1} , P_{m-2} is aligned under T_{n-2} , etc. Thus, P_1 , which can be written as $P_{m-(m-1)}$, is aligned under $T_{n-(m-1)}$, which is T_{n-m+1} . 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 2.16

Final Draft of the Pattern-Matching Algorithm

Pattern-Matching Algorithm

```

Get values for  $n$  and  $m$ , the size of the text and the pattern, respectively
Get values for both the text  $T_1 T_2 \dots T_n$  and the pattern  $P_1 P_2 \dots P_m$ 
Set  $k$ , the starting location for the attempted match, to 1
While ( $k \leq (n - m + 1)$ ) do
  Set the value of  $i$  to 1
  Set the value of Mismatch to NO
  While both ( $i \leq m$ ) and (Mismatch = NO) do
    If  $P_i \neq T_{k+(i-1)}$  then
      Set Mismatch to YES
    Else
      Increment  $i$  by 1 (to move to the next character)
  End of the loop
  If Mismatch = NO then
    Print the message ‘There is a match at position’
    Print the value of  $k$ 
  Increment  $k$  by 1
End of the loop
Stop, we are finished

```

PRACTICE PROBLEMS

1. Consider the following “telephone book.”

<i>Name</i>	<i>Number</i>
Smith	555-1212
Jones	834-6543
Adams	921-5281
Doe	327-8900

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

- Adams
 - Schneider
2. 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.

3. Consider the following text.

Text: A man and a woman

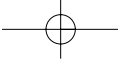
Trace the pattern-matching algorithm of Figure 2.16 using the 2-character pattern ‘an’ and show the output produced.

4. 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.

2.4 Conclusion

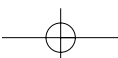
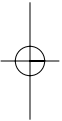
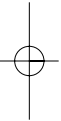
You have now had a chance to see the step-by-step design and development of some interesting, nontrivial algorithms. You have also been introduced to a number of important concepts related to problem solving, including algorithm design and discovery, pseudocode, control statements, iteration, libraries, 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 solution.

Designing a technically correct algorithm to solve a given problem is only part of what computer scientists do. They also must 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 directory assistance program that takes 10 minutes to locate a telephone number would be of little or no use. A caller would surely hang up long before the answer was found. 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.



EXERCISES

1. Write pseudocode instructions to carry out each of the following computational operations.
 - a. Determine the area of a triangle given values for the base b and the height h .
 - b. Compute the interest earned in 1 year given the starting account balance B and the annual interest rate I and assuming simple interest, that is, no compounding. Also determine the final balance at the end of the year.
 - c. Determine the flying time between two cities given the mileage M between them and the average speed of the airplane.
2. Using only the sequential operations described in Section 2.2.2, write an algorithm that gets values for the starting account balance B , annual interest rate I , and annual service charge S . Your algorithm should output the amount of interest earned during the year and the final account balance at the end of the year. Assume that interest is compounded monthly and the service charge is deducted once, at the end of the year.
3. Using only the sequential operations described in Section 2.2.2, write an algorithm that gets four numbers corresponding to scores received on three semester tests and a final examination. Your algorithm should compute and display the average of all four tests, weighting the final exam twice as heavily as a regular test.
4. Write an algorithm that gets the price for item A plus the quantity purchased. The algorithm prints the total cost, including a 6% sales tax.
5. Write an if/then/else primitive to do each of the following operations.
 - a. Compute and display the value $x \div y$ if the value of y is not 0. If y does have the value 0, then display the message 'Unable to perform the division.'
 - b. Compute the area and circumference of a circle given the radius r if the radius is greater than or equal to 1.0; otherwise, you should compute only the circumference.
6. Modify the algorithm of Exercise 2 to include the annual service charge only if the starting account balance at the beginning of the year is less than \$1,000. If it is greater than or equal to \$1,000, then there is no annual service charge.
7. Write an algorithm that uses a loop (1) to input 10 pairs of numbers, where each pair represents the score of a football game with the Computer State University (CSU) score listed first, and (2) for each pair of numbers, determine whether CSU won or lost. After reading in these 10 pairs of values, print out the won/lost/tie record of CSU. In addition, if this record is a perfect 10-0, then print out the message 'Congratulations on your undefeated season.'
8. Modify the test-averaging algorithm of Exercise 3 so that it reads in 15 test scores rather than 4. There are 14 regular tests and a final examination, which counts twice as much as a regular test. Use a loop to input and sum the scores.
9. Modify the sales computation algorithm of Exercise 4 so that after finishing the computation for one item, it starts on the computation for the next. This iterative process is repeated until the total cost exceeds \$1000.
10. Write an algorithm that is given your electric meter readings (in kilowatt-hours) at the beginning and end of each month of the year. The algorithm determines your annual cost of electricity on the basis of a charge of 6 cents per kilowatt-hour for the first 1,000 kilowatt-hours of each month and 8 cents per kilowatt-hour beyond 1,000. After printing out your total annual charge, the algorithm also determines whether you used less than 500 kilowatt-hours for the entire year and, if so, prints out a message thanking you for conserving electricity.
11. Develop an algorithm to compute gross pay. The inputs to your algorithm are the hours worked per week and the hourly pay rate. The rule for determining gross pay is to pay the regular pay rate for all hours worked up to 40, time-and-a-half for all hours over 40 up to 54, and double time for all hours over 54. Compute and display the value for gross pay using this rule. After displaying one value, ask the user whether he or she wants to do another computation. Repeat the entire set of operations until the user says no.
12. Develop a formal argument that "proves" that the sequential search algorithm shown in Figure 2.13 cannot have an infinite loop; that is, prove that it will always stop after a finite number of operations.
13. Modify the sequential search algorithm of Figure 2.13 so that it works correctly even if the names in the directory are not unique, that is, if the desired name occurs more than once. Your modified algorithm should find every occurrence of *NAME* in the directory and print out the telephone number corresponding to every match. In addition, after all the numbers have been displayed, your algorithm should print out how many occurrences of *NAME* were located. For example, if *NAME* occurred three times, the output of the algorithm might look something like this:


```
528-5638
922-7874
488-2020
```

A total of three occurrences were located.

14. Use the Find Largest algorithm of Figure 2.14 to help you develop an algorithm to find the median value in a list containing N unique numbers. The median of N numbers is defined as the value in the list in which approximately half the values are larger than it and half the values are smaller than it. For example, consider the following list of seven numbers.

26, 50, 83, 44, 91, 20, 55

The median value is 50 because three values (20, 26, and 44) are smaller and three values (55, 83, and 91) are larger. If N is an even value, then the number of values larger than the median will be one greater than the number of values smaller than the median.

15. With regard to the Find Largest algorithm of Figure 2.14, if the numbers in our list were not unique and therefore the largest number could occur more than once, would the algorithm find the first occurrence? The last occurrence? Every occurrence? Explain precisely how this algorithm would behave when presented with this new condition.
16. On the sixth line of the Find Largest algorithm of Figure 2.14 there is an instruction that reads,

While ($i \leq n$) do

Explain exactly what would happen if we changed that instruction to read as follows:

- While ($i \geq n$) do
- While ($i < n$) do
- While ($i = n$) do

17. On the seventh line of the Find Largest algorithm of Figure 2.14 is an instruction that reads,

If $A_i >$ largest so far then . . .

Explain exactly what would happen if we changed that instruction to read as follows:

- If $A_i \geq$ largest so far then . . .
- If $A_i <$ largest so far then . . .

Looking back over your answers to the previous two questions, what do they say about the importance of using the correct *relational operation* ($<$, $=$, $>$, \geq , \leq , \neq) when writing out either an iterative or conditional algorithmic primitive?

18. a. Refer to the pattern-matching algorithm in Figure 2.16. What is the output of the algorithm as it currently stands if our text is

Text: We must band together and handle adversity and we search for the pattern “and”?

- b. How could we modify the algorithm so that it finds only the complete word *and* rather than the occurrence of the character sequence *a, n,* and *d* that are contained within another word, such as *band*?

19. Refer to the pattern-matching algorithm in Figure 2.16. Explain how the algorithm would behave if we accidentally omitted the statement on line 16 that says,

Increment k by 1

20. Design an algorithm that is given a positive integer N and determines whether N is a prime number, that is, not evenly divisible by any value other than 1 and itself. The output of your algorithm is either the message “not prime,” along with a factor of N , or the message “prime.”

21. Write an algorithm that generates a Caesar cipher—a secret message in which each letter is replaced by the one that is k letters ahead of it in the alphabet, in a circular fashion. For example, if $k = 5$, then the letter *a* would be replaced by the letter *f*, and the letter *x* would be replaced by the letter *c*. (We’ll talk more about the Caesar cipher and other encryption algorithms in Chapter 13.) The input to your algorithm is the text to be encoded, ending with the special symbol “\$”, and the value k . (You may assume that, except for the special ending character, the text contains only the 26 letters *a . . . z*.) The output of your algorithm is the encoded text.

22. Design and implement an algorithm that is given as input an integer value $k \geq 0$ and a list of k numbers N_1, N_2, \dots, N_k . Your algorithm should reverse the order of the numbers in the list. That is, if the original list contained:

$N_1 = 5, N_2 = 13, N_3 = 8, N_4 = 27, N_5 = 10$ ($k = 5$)

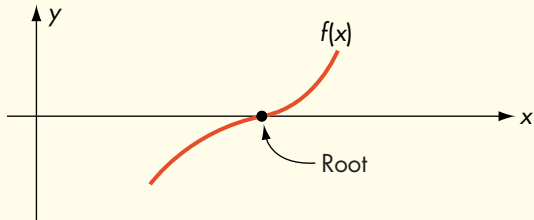
then when your algorithm has completed, the values stored in the list will be:

$N_1 = 10, N_2 = 27, N_3 = 8, N_4 = 13, N_5 = 5$

23. Design and implement an algorithm that gets as input a list of k integer values N_1, N_2, \dots, N_k as well as a special value SUM . Your algorithm must locate a pair of values in the list N that sum to the value SUM . For example, if your list of values is 3, 8, 13, 2, 17, 18, 10, and the value of SUM is 20, then your algorithm would output either the two values (2, 18) or (3, 17). If your algorithm cannot find any pair of values that sum to the value SUM , then it should print out the message ‘Sorry, there is no such pair of values.’

CHALLENGE WORK

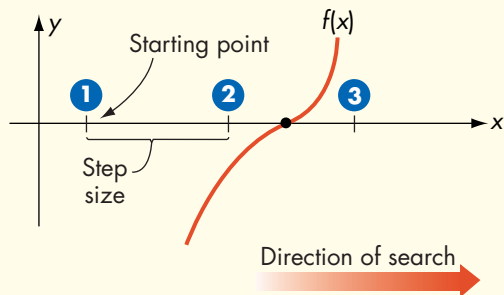
- Design an algorithm to find the *root* of a function $f(x)$, where the root is defined as a point x such that $f(x) = 0$. Pictorially, the root of a function is the point where the graph of that function crosses the x -axis.



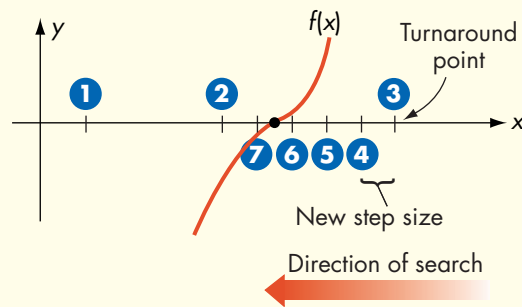
Your algorithm should operate as follows. Initially it will be given three values:

- A starting point for the search
- A step size
- The accuracy desired

Your algorithm should begin at the specified starting point and begin to “walk up” the x -axis in units of step size. After taking a step, it should ask the question “Have I passed a root?” It can determine the answer to this question by seeing whether the sign of the function has changed from the previous point to the current point. (Note that below the axis the sign of $f(x)$ is negative; above the axis it is positive. If it crosses the x -axis, it must change sign.) If the algorithm has not passed a root, it should keep walking up the x -axis until it does. Pictorially,



When the algorithm passes a root, it must do two things. First, it changes the sign of step size so that it starts walking in the reverse direction, because it is now past the root. Second, it multiplies step size by 0.1, so our steps are 1/10 as big as they were before. We now repeat the operation described above, walking down the x -axis until we pass the root.



Again, the algorithm changes the sign of step size to reverse direction and reduces it to 1/10 its previous size. As the diagrams show, we are slowly zeroing in on the root—going past it, turning around, going past it, turning around, and so forth. This iterative process stops when the algorithm passes a root and the step size is smaller than the desired accuracy. It has now bracketed the root within an interval that is smaller than the accuracy we want. At this point it should print out the midpoint of the interval and terminate.

There are many special cases that this algorithm must deal with, but in your solution you may disregard them. Assume that you will always encounter a root in your “travels” along the x -axis. After creating a solution, you may wish to look at some of these special cases, such as a function that has no real roots, a starting point that is to the right of all the roots, and two roots so close together that they fall within the same step.

- One of the most important and widely used classes of algorithms in computer science is **sorting**, the process of putting a list of elements into a predefined order, usually numeric or alphabetic. There are many different sorting algorithms, and we will look at some of them in Chapter 3. One of the simplest sorting algorithms is called **selection sort**, and it can be implemented using the tools that you have learned in this chapter. It is also one of the easiest to understand as it mimics how we often sort collections of values when we must do it by hand.

Assume that we are given a list named A , containing eight values that we want to sort into ascending order, from smallest to largest:

$A:$ 23 18 66 9 21 90 32 4
 Position: 1 2 3 4 5 6 7 8

We first look for the largest value contained in positions 1 to 8 of list A . We can do this using something like the

Find Largest algorithm that appears in Figure 2.14. In this case the largest value is 90, and it appears in position 6. Since this is the largest value in A, we swap it with the value in position 8 so that it is in its correct place at the back of the list. The list is now partially sorted from position 8 to position 8:

A: 23 18 66 9 21 4 32 90
 Position: 1 2 3 4 5 6 7 8

We now search the array for the second largest value. Since we know that the largest value is contained in position 8, we need to search only positions 1 to 7 of list A to find the second largest value. In this case the second largest value is 66, and it appears in position 3. We now swap the value in position 3 with the value in position 7 to get the second largest value into its correct location. This produces the following:

A: 23 18 32 9 21 4 66 90
 Position: 1 2 3 4 5 6 7 8

The list is now partially sorted from position 7 to position 8, with those two locations holding the two largest values. The next search goes from position 1 to position 6 of list A, this time trying to locate the third largest value, and we swap that value with the number in position 6. After repeating

this process 7 times, the list is completely sorted. (That is because if the last 7 items are in their correct place, the item in position 1 must also be in its correct place.)

Using the Find Largest algorithm shown in Figure 2.14 (which may have to be slightly modified) and the primitive pseudocode operations listed in Figure 2.9, implement the selection sort algorithm that we have just described. Assume that n , the size of the list, and the n -element list A are input to your algorithm. The output of your algorithm should be the sorted list.

- Most people are familiar with the work of the great mathematicians of ancient Greece and Rome, such as Archimedes, Euclid, Pythagoras, and Plato. However, a great deal of important work in arithmetic, geometry, algebra, number theory, and logic was carried out by scholars working in Egypt, Persia, India, and China. For example, the concept of zero was first developed in India, and positional numbering systems (like our own decimal system) were developed and used in China, India, and the Middle East long before they made their way to Europe. Read about the work of some mathematician (such as Al-Khwarizmi) from these or other places, and write a paper describing his or her contributions to mathematics, logic, and (ultimately) computer science.

FOR FURTHER READING

The classic text on algorithms is the three-volume series by Donald Knuth:

Knuth, D. *The Art of Computer Programming*, 3 vols. Reading, MA: Addison Wesley, 1997–1998.

Volume 1: *Fundamental Algorithms*, 3rd ed., 1997.

Volume 2: *Seminumerical Algorithms*, 3rd ed., 1998.

Volume 3: *Sorting and Searching*, 2nd ed., 1998.

The following books provide additional information about the design of algorithms to solve a wide range of interesting problems.

Baase, S., and Van Gelder, A., *Computer Algorithms: Introduction to Design and Analysis*, 3rd ed. Reading, MA: Addison Wesley, 2000.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. *Introduction to Algorithms*, 2nd ed. New York: McGraw-Hill, 2002.

Harel, D. *Algorithmics: The Spirit of Computing*, 2nd ed. Reading, MA: Addison Wesley, 1992.

Michalewicz, Z., and Fogel, D. *How to Solve It: Modern Heuristics*. Amsterdam: Springer-Verlag, 1999.

Skiena, S. *The Algorithm Design Manual*. New York: Telos Press, 1997.

The following is an excellent introduction to algorithm design using the control of the motions and actions of a toy robot as a basis for teaching algorithmic problem solving.

Pattis, R.; Roberts, J.; and Stehlik, M. *Karel the Robot: A Gentle Introduction to the Art of Programming*, 2nd ed. New York: Wiley, 1994.

