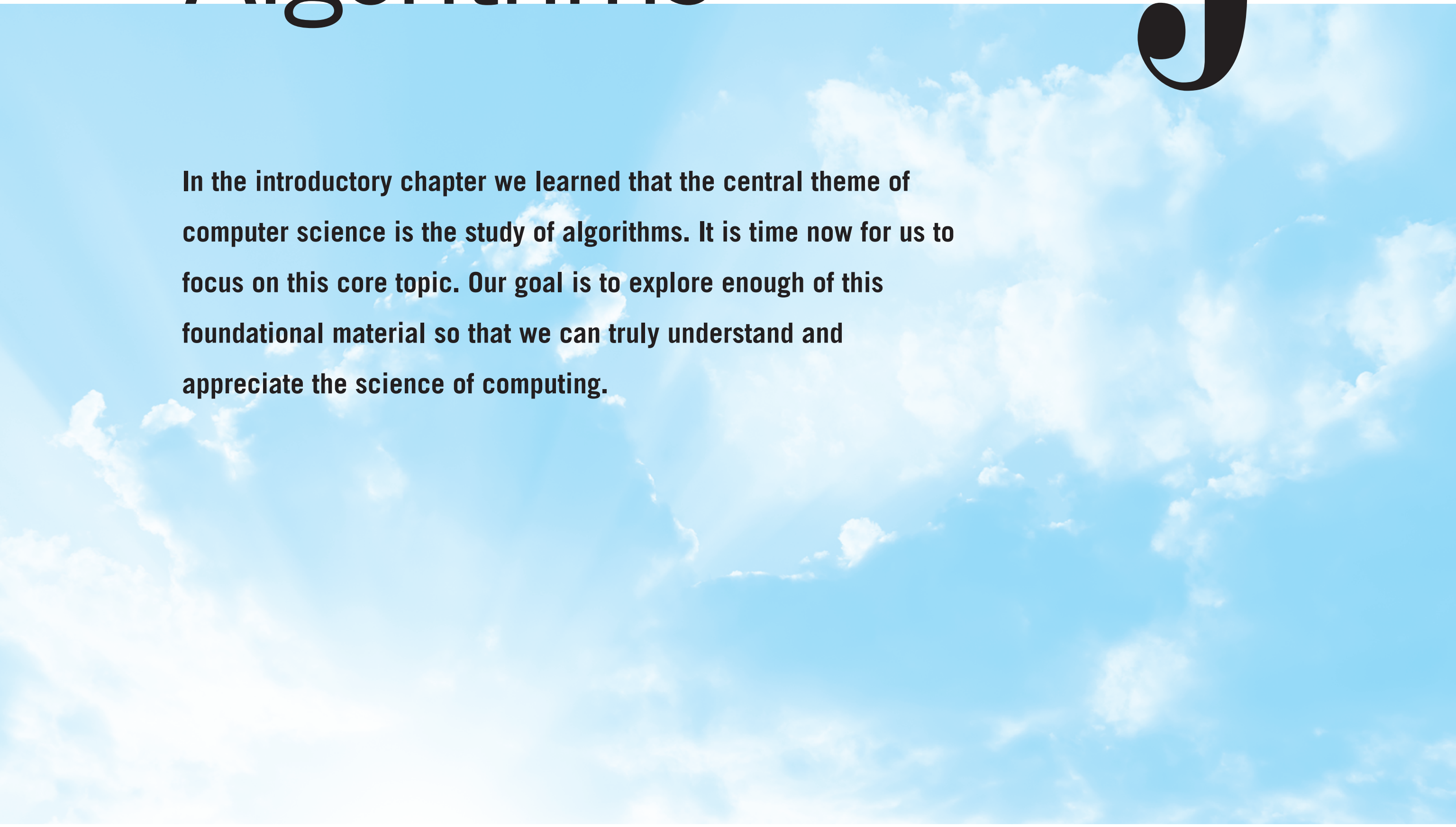
CHAPTER 5

Algorithms



In the introductory chapter we learned that the central theme of computer science is the study of algorithms. It is time now for us to focus on this core topic. Our goal is to explore enough of this foundational material so that we can truly understand and appreciate the science of computing.

We have seen that before a computer can perform a task, it must be given an algorithm telling it precisely what to do; consequently, the study of algorithms is the cornerstone of computer science. In this chapter we introduce many of the fundamental concepts of this study, including the issues of algorithm discovery and representation as well as the major control concepts of iteration and recursion. In so doing we also present a few well-known algorithms for searching and sorting. We begin by reviewing the concept of an algorithm.

5.1 The Concept of an Algorithm

In the introductory chapter we informally defined an algorithm as a set of steps that define how a task is performed. In this section we look more closely at this fundamental concept.

An Informal Review

We have encountered a multitude of algorithms in our study. We have found algorithms for converting numeric representations from one form to another, detecting and correcting errors in data, compressing and decompressing data files, controlling multiprogramming in a multitasking environment, and many more. Moreover, we have seen that the machine cycle that is followed by a CPU is nothing more than the simple algorithm

```
As long as the halt instruction has not been executed continue to
execute the following steps:
  a. Fetch an instruction.
  b. Decode the instruction.
  c. Execute the instruction.
```

As demonstrated by the algorithm describing a magic trick in Figure 0.1, algorithms are not restricted to technical activities. Indeed, they underlie even such mundane activities as shelling peas:

```
Obtain a basket of unshelled peas and an empty bowl. As long as
there are unshelled peas in the basket continue to execute the
following steps:
  a. Take a pea from the basket.
  b. Break open the pea pod.
  c. Dump the peas from the pod into the bowl.
  d. Discard the pod.
```

In fact, many researchers believe that every activity of the human mind, including imagination, creativity, and decision making, is actually the result of algorithm execution—a conjecture we will revisit in our study of artificial intelligence (Chapter 11).

But before we proceed further, let us consider the formal definition of an algorithm.

The Formal Definition of an Algorithm

Informal, loosely defined concepts are acceptable and common in everyday life, but a science must be based on well-defined terminology. Consider, then, the formal definition of an algorithm stated in Figure 5.1.

Figure 5.1 The definition of an algorithm

An algorithm is an ordered set
of unambiguous, executable steps
that defines a terminating process.

Note that the definition requires that the set of steps in an algorithm be ordered. This means that the steps in an algorithm must have a well-established structure in terms of the order of their execution. This does not mean, however, that the steps must be executed in a sequence consisting of a first step, followed by a second, and so on. Some algorithms, known as parallel algorithms, contain more than one sequence of steps, each designed to be executed by different processors in a multiprocessor machine. In such cases the overall algorithm does not possess a single thread of steps that conforms to the first-step, second-step scenario. Instead, the algorithm's structure is that of multiple threads that branch and reconnect as different processors perform different parts of the overall task. (We will revisit this concept in Chapter 6.) Other examples include algorithms executed by circuits such as the flip-flop in Chapter 1, in which each gate performs a single step of the overall algorithm. Here the steps are ordered by cause and effect, as the action of each gate propagates throughout the circuit.

Next, consider the requirement that an algorithm must consist of executable steps. To appreciate this condition, consider the instruction

```
Make a list of all the positive integers
```

which would be impossible to perform because there are infinitely many positive integers. Thus any set of instructions involving this instruction would not be an algorithm. Computer scientists use the term *effective* to capture the concept of being executable. That is, to say that a step is effective means that it is doable.

Another requirement imposed by the definition in Figure 5.1 is that the steps in an algorithm be unambiguous. This means that during execution of an algorithm, the information in the state of the process must be sufficient to determine uniquely and completely the actions required by each step. In other words, the execution of each step in an algorithm does not require creative skills. Rather, it requires only the ability to follow directions. (In Chapter 12 we will learn that "algorithms," called nondeterministic algorithms, that do not conform to this restriction are an important topic of research.)

The definition in Figure 5.1 also requires that an algorithm define a terminating process, which means that the execution of an algorithm must lead to an end. The origin of this requirement is in theoretical computer science, where the goal is to answer such questions as "What are the ultimate limitations of algorithms and machines?" Here computer science seeks to distinguish between problems whose answers can be obtained algorithmically and problems whose answers lie beyond the capabilities of algorithmic systems. In this context, a line is drawn between processes that culminate with an answer and those that merely proceed forever without producing a result.

There are, however, meaningful applications for nonterminating processes, including monitoring the vital signs of a hospital patient and maintaining an aircraft's altitude in flight. Some would argue that these applications involve merely the repetition of algorithms, each of which reaches an end and then automatically repeats. Others would counter that such arguments are simply attempts to cling to an overly restrictive formal definition. In any case, the result is that the term *algorithm* is often used in applied, or informal, settings in reference to sets of steps that do not necessarily define terminating processes. An example is the long-division "algorithm" that does not define a terminating process for dividing 1 by 3. Technically, such instances represent misuses of the term.

The Abstract Nature of Algorithms

It is important to emphasize the distinction between an algorithm and its representation—a distinction that is analogous to that between a story and a book. A story is abstract, or conceptual, in nature; a book is a physical representation of a story. If a book is translated into another language or republished in a different format, it is merely the representation of the story that changes—the story itself remains the same.

In the same manner, an algorithm is abstract and distinct from its representation. A single algorithm can be represented in many ways. As an example, the algorithm for converting temperature readings from Celsius to Fahrenheit is traditionally represented as the algebraic formula

$$F = (9/5)C + 32$$

But it could be represented by the instruction

Multiply the temperature reading in Celsius by $9/5$
and then add 32 to the product

or even in the form of an electronic circuit. In each case the underlying algorithm is the same; only the representations differ.

The distinction between an algorithm and its representation presents a problem when we try to communicate algorithms. A common example involves the level of detail at which an algorithm must be described. Among meteorologists, the instruction "Convert the Celsius reading to its Fahrenheit equivalent" suffices, but a layperson, requiring a more detailed description, might argue that the instruction is ambiguous. The problem, however, is not with the underlying algorithm but that the algorithm is not represented in enough detail for the layperson. In the next section we will see how the concept of primitives can be used to eliminate such ambiguity problems in an algorithm's representation.

Finally, while on the subject of algorithms and their representations, we should clarify the distinction between two other related concepts—programs and processes. A *program* is a representation of an algorithm. (Here we are using the term *algorithm* in its less formal sense in that many programs are representations of nonterminating "algorithms.") In fact, within the computing community the term *program* usually refers to a formal representation of an algorithm designed for computer application. We defined a *process* in Chapter 3 to be the activity of executing a program. Note, however, that to execute a program is to execute the algorithm represented by the program, so a process could equivalently be defined as the activity of executing an algorithm. We conclude that programs, algorithms, and processes are distinct, yet related, entities. A program is the representation of an algorithm, whereas a process is the activity of executing an algorithm.

Questions & Exercises

1. Summarize the distinctions between a process, an algorithm, and a program.
2. Give some examples of algorithms with which you are familiar. Are they really algorithms in the precise sense?
3. Identify some points of vagueness in our informal definition of an algorithm introduced in Section 0.1 of the introductory chapter.
4. In what sense do the steps described by the following list of instructions fail to constitute an algorithm?

   Step 1. Take a coin out of your pocket and put it on the table.

   Step 2. Return to Step 1.

5.2 Algorithm Representation

In this section we consider issues relating to an algorithm's representation. Our goal is to introduce the basic concepts of primitives and pseudocode as well as to establish a representation system for our own use.

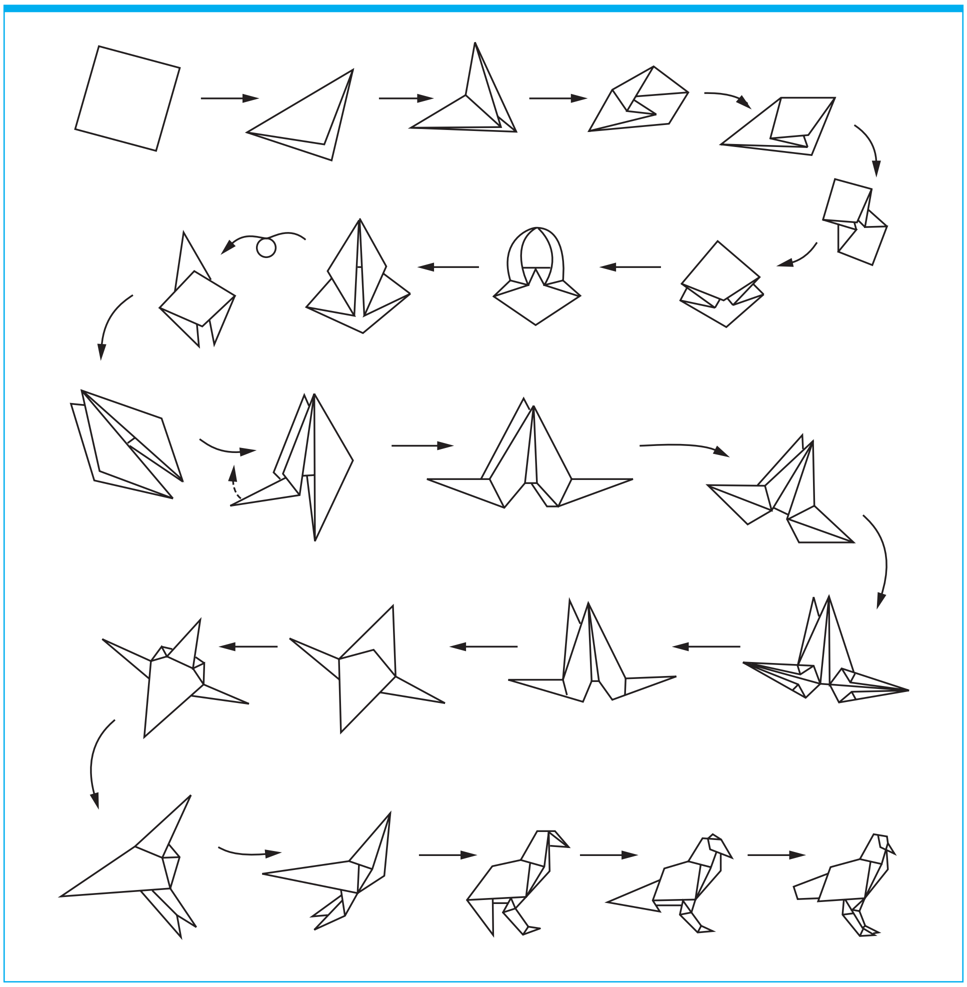
Primitives

The representation of an algorithm requires some form of language. In the case of humans this might be a traditional natural language (English, Spanish, Russian, Japanese) or perhaps the language of pictures, as demonstrated in Figure 5.2, which describes an algorithm for folding a bird from a square piece of paper. Often, however, such natural channels of communication lead to misunderstandings, sometimes because the terminology used has more than one meaning. (The sentence, "Visiting grandchildren can be nerve-racking," could mean either that the grandchildren cause problems when they come to visit or that going to see them is problematic.) Problems also arise over misunderstandings regarding the level of detail required. Few readers could successfully fold a bird from the directions given in Figure 5.2, yet a student of origami would probably have little difficulty. In short, communication problems arise when the language used for an algorithm's representation is not precisely defined or when information is not given in adequate detail.

Computer science approaches these problems by establishing a well-defined set of building blocks from which algorithm representations can be constructed. Such a building block is called a **primitive.** Assigning precise definitions to these primitives removes many problems of ambiguity, and requiring algorithms to be described in terms of these primitives establishes a uniform level of detail. A collection of primitives along with a collection of rules stating how the primitives can be combined to represent more complex ideas constitutes a **programming language.**

Each primitive has its own syntax and semantics. Syntax refers to the primitive's symbolic representation; semantics refers to the meaning of the primitive. The syntax of *air* consists of three symbols, whereas the semantics is a gaseous substance that surrounds the world. As an example, Figure 5.3 presents some of the primitives used in origami.

Figure 5.2 Folding a bird from a square piece of paper



To obtain a collection of primitives to use in representing algorithms for computer execution, we could turn to the individual instructions that the machine is designed to execute. If an algorithm is expressed at this level of detail, we will certainly have a program suitable for machine execution. However, expressing algorithms at this level is tedious, and so one normally uses a collection of "higher-level" primitives, each being an abstract tool constructed from the lower-level primitives provided in the machine's language. The result is a formal programming language in which algorithms can be expressed at a conceptually higher level than in machine language. We will discuss such programming languages in the next chapter.

Pseudocode

For now, we forgo the introduction of a formal programming language in favor of a less formal, more intuitive notational system known as pseudocode. In general, a **pseudocode** is a notational system in which ideas can be expressed informally during the algorithm development process.

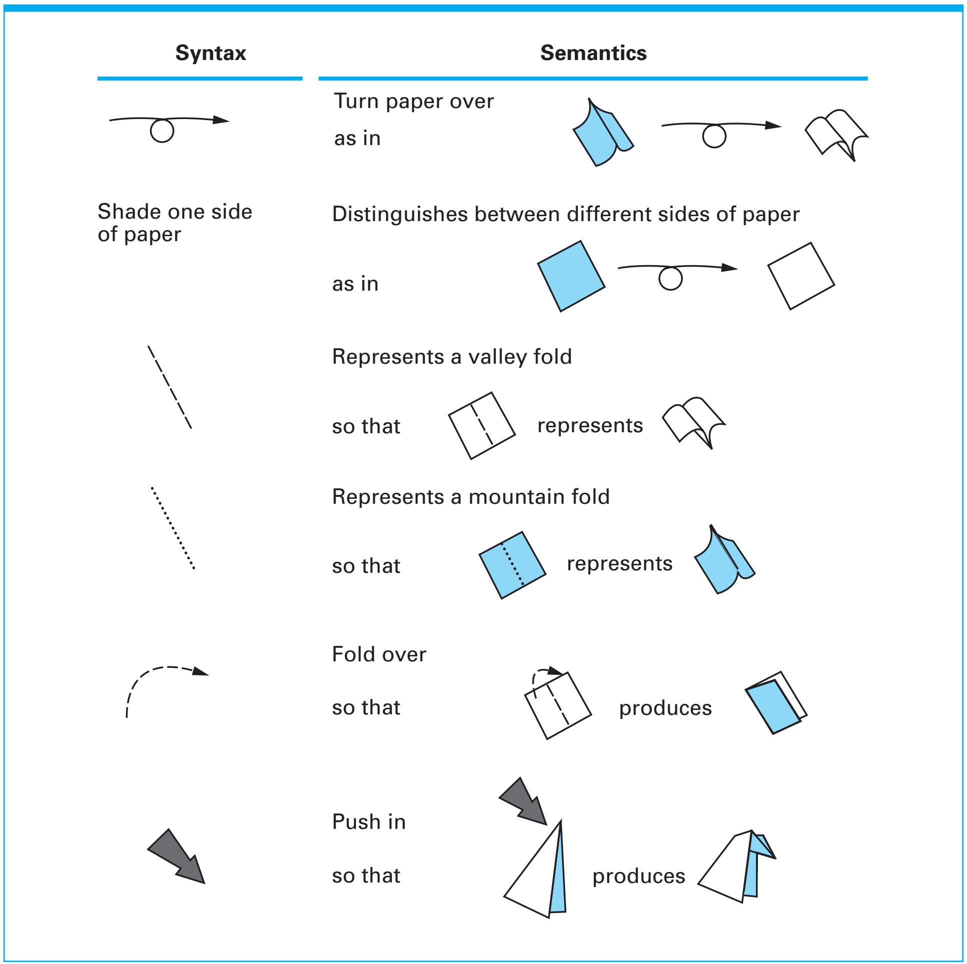


Figure 5.3 Origami primitives

One way to obtain a pseudocode is simply to loosen the rules of a formal programming language, borrowing the syntax-semantic structures of the language, intermixed with less formal constucts. There are many such pseudocode variants, because there are many programming languages in existence. Two particularly popular choices are loose versions of the languages Algol and Pascal, largely because these were widely used in textbooks and academic papers for decades. More recently, pseudocode reminiscent of the Java and C languages has proliferated, again because most programmers will have at least a reading knowledge of these languages. Regardless of where a pseudocode borrows its syntax from, one essential property is required for it to serve its purpose in expressing algorithms: A pseudocode must have a consistent, concise notation for representing recurring semantic structures. For our purposes, we will use a Python-like syntax to express pseudocode throughout the remainder of the book. Some of our pseudocode semantic structures will borrow from language constructs presented in previous chapters, while others will look ahead to constructs to be formally covered in future chapters.

One such recurring semantic structure is the saving of a computed value. For example, if we have computed the sum of our checking and savings account

Algorithm Representation During Algorithm Design

The task of designing a complex algorithm requires that the designer keep track of numerous interrelated concepts—a requirement that can exceed the capabilities of the human mind. Thus the designer of complex algorithms needs a way to record and recall portions of an evolving algorithm as his or her concentration requires.

During the 1950s and 1960s, flowcharts (by which algorithms are represented by geometric shapes connected by arrows) were the state-of-the-art design tool. However, flowcharts often became tangled webs of crisscrossing arrows that made understanding the structure of the underlying algorithm difficult. Thus the use of flowcharts as design tools has given way to other representation techniques. An example is the pseudocode used in this text, by which algorithms are represented with well-defined textual structures. Flowcharts are still beneficial when the goal is presentation rather than design. For example, Figures 5.8 and 5.9 apply flowchart notation to demonstrate the algorithmic structure represented by popular control statements.

The search for better design notations is a continuing process. In Chapter 7 we will see that the trend is to use graphical techniques to assist in the global design of large software systems, while pseudocode remains popular for designing the smaller procedural components within a system.

balances, we may want to save the result so we can refer to it later. In such cases we will use the form

```
name = expression
```

where `name` is the name by which we will refer to the result and `expression` describes the computation whose result is to be saved. This pseudocode structure directly follows the equivalent Python **assignment statement,** which we introduced in Chapter 1 for storing a value into a Python variable. For example, the statement

```
RemainingFunds = CheckingBalance + SavingsBalance
```

is an assignment statement that assigns the sum of `CheckingBalance` and `SavingsBalance` to the name `RemainingFunds`. Thus, the term `RemainingFunds` can be used in future statements to refer to that sum.

Another recurring semantic structure is the selection of one of two possible activities depending on the truth or falseness of some condition. Examples include:

> If the gross domestic product has increased, buy common stock; otherwise, sell common stock.
>
> Buy common stock if the gross domestic product has increased and sell it otherwise.
>
> Buy or sell common stock depending on whether the gross domestic product has increased or decreased, respectively.

Each of these statements could be rewritten to conform to the structure

```
if (condition):
  activity
else:
  activity
```

From Pseudocode to Python

Our pseudocode in this chapter closely mirrors actual Python syntax for the `if` and `while` structures, as well as function definition and calling syntax.

```
if (condition):
  activity
else:
  activity
```

Our pseudocode `if` and `while` structures can be converted to Python simply by being more precise with the **`condition`** and **`activity`** portions. For example, rather than the English phrase, “sales have decreased” as a condition, we would need a proper Python comparison expression, such as

```
if (sales_current < sales_previous):
```

where the sales variables had already been assigned in previous lines of the script. Similarly, informal English sentences or phrases used as the **`activity`** in our pseudocode would need to be replaced with Python statements and expressions such as those we have already seen in earlier chapters.

How can you tell the difference between pseudocode and actual Python code in this book? As a rule, real Python code uses operators (like “<”, “=”, or “+”) to string together multiple named variables into more complex expressions, or commas to separate a list of parameters being sent to a function. Articles like “the” and “a,” or prepositions like “from” appear in pseudocode. We use periods at the end of sentences in pseudocode; Python statements do not have punctuation at the end.

where we have used the keywords `if` and `else` to announce the different substructures within the main structure and have used colons and indentation to delineate the boundaries of these substructures. The **`condition`** and the `else` will always be followed immediately by a colon. The corresponding **`activity`** will be indented. If an **`activity`** consists of multiple steps, they will all be similarly indented. By adopting this syntactic structure for our pseudocode, we acquire a uniform way in which to express this common semantic structure. Thus, whereas the statement

> Depending on whether the year is a leap year, divide the total by 366 or 365, respectively

might possess a more creative literary style, we will consistently opt for the straightforward

```
if (year is leap year):
  daily total = total / 366
else:
  daily total = total / 365
```

We also adopt the shorter syntax

```
if (condition):
  Activity
```

for those cases not involving an else activity. Using this notation, the statement

Should it be the case that sales have decreased, lower the price by 5%.

will be reduced to

```
if (sales have decreased):
  lower the price by 5%
```

Still another common semantic structure is the repeated execution of a statement or sequence of statements as long as some condition remains true. Informal examples include

As long as there are tickets to sell, continue selling tickets.

and

While there are tickets to sell, keep selling tickets.

For such cases, we adopt the uniform pattern

```
while (condition):
  Activity
```

for our pseudocode. In short, such a statement means to check the `condition` and, if it is true, perform the `activity` and return to check the `condition` again. If, however, the `condition` is found to be false, move on to the next instruction following the `while` structure. Thus both of the preceding statements are reduced to

```
while (tickets remain to be sold):
  sell a ticket
```

In many programming languages, indentation often enhances the readability of a program. In Python, and thus also in our Python-derived pseudocode, indentation is essential to the notation. For example, in the statement

```
if (not raining):
  if (temperature == hot):
    go swimming
  else:
    play golf

else:
  watch television
```

indentation tells us that the question of whether `temperature` equals `hot` will not even be asked unless it is `not raining`. Note the use of double equal signs, "==" to differentiate between assignment (=) and comparison (==). The question about the temperature is **nested** inside the if-statement, and is, in effect, the activity to be performed if the outer if-statement condition holds true. Similarly, indentation tells us that `else: play golf` belongs to the inner if-statement, rather than to the outer if-statement. Thus we will adopt the use of indentation in our pseudocode.

We want to use our pseudocode to describe activities that can be used as abstract tools in other applications. Computer science has a variety of terms for such program units, including subprogram, subroutine, procedure, method, and function, each with its own variation of meaning. We will follow Python convention, using the term **function** for our pseudocode and using the Python keyword def to announce the title by which the pseudocode unit will be known. More precisely, we will begin a pseudocode unit with a statement of the form

```
def name():
```

where name is the particular name of the unit. We will then follow this introductory statement with the statements that define the unit's action. For example, Figure 5.4 is a pseudocode representation of a function called Greetings that prints the message "Hello" three times.

When the task performed by a function is required elsewhere in our pseudocode, we will merely request it by name. For example, if two functions were named ProcessLoan and RejectApplication, then we could request their services within an if-else structure by writing

```
if (. . . ):
  ProcessLoan()
else:
  RejectApplication()
```

which would result in the execution of the function ProcessLoan if the tested condition were true or in the execution of RejectApplication if the condition were false.

If functions are to be used in different situations, they should be designed to be as generic as possible. A function for sorting lists of names should be designed to sort any list—not a particular list—so it should be written in such a way that the list to be sorted is not specified in the function itself. Instead, the list should be referred to by a generic name within the function's representation.

In our pseudocode, we will adopt the convention of listing these generic names (which are called **parameters**) in parentheses on the same line on which we identify the function's name. In particular, a function named Sort, which is designed to sort any list of names, would begin with the statement

```
def Sort (List):
```

Later in the representation where a reference to the list being sorted is required, the generic name List would be used. In turn, when the services of Sort are required, we will identify which list is to be substituted for List in the function Sort. Thus we will write something such as

```
Sort(the organization's membership list)
```

and

```
Sort(the wedding guest list)
```

depending on our needs.

Figure 5.4 The function Greetings in pseudocode

```
def Greetings():
  Count = 3
  while (Count > 0):
    print('Hello')
    Count = Count - 1
```

Naming Items in Programs

In a natural language, items often have multiword names such as "cost of producing a widget" or "estimated arrival time." Experience has shown that use of such multiword names in the representation of an algorithm can complicate the algorithm's description. It is better to have each item identified by a single contiguous block of text. Over the years many techniques have been used to compress multiple words into a single lexical unit to obtain descriptive names for items in programs. One is to use underlines to connect words, producing names such as `estimated_arrival_time`. Another is to use uppercase letters to help a reader comprehend a compressed multiword name. For example, one could start each word with an uppercase letter to obtain names such as `EstimatedArrivalTime`. This technique is called **Pascal casing,** because it was popularized by users of the Pascal programming language. A variation of Pascal casing is called **camel casing,** which is identical to Pascal casing except that the first letter remains in lowercase as in `estimatedArrivalTime`. In this text we lean toward Pascal casing, but the choice is largely a matter of taste.

Questions & Exercises

1. A primitive in one context might turn out to be a composite of primitives in another. For instance, our `while` statement is a primitive in our pseudocode, yet it is ultimately implemented as a composite of machine-language instructions. Give two examples of this phenomenon in a non-computer setting.
2. In what sense is the construction of functions the construction of primitives?
3. The Euclidean algorithm finds the greatest common divisor of two positive integers X and Y by the following process:

   As long as the value of neither X nor Y is zero, assign the larger the remainder of dividing the larger by the smaller. The greatest common divisor, if it exists, will be the remaining non-zero value.

   Express this algorithm in our pseudocode.
4. Describe a collection of primitives that are used in a subject other than computer programming.

5.3 Algorithm Discovery

The development of a program consists of two activities—discovering the underlying algorithm and representing that algorithm as a program. Up to this point we have been concerned with the issues of algorithm representation without considering the question of how algorithms are discovered in the first place. Yet algorithm discovery is usually the more challenging step in the software development process. After all, discovering an algorithm to solve a problem requires finding a method of solving that problem. Thus, to understand how algorithms are discovered is to understand the problem-solving process.

The Art of Problem Solving

The techniques of problem solving and the need to learn more about them are not unique to computer science but rather are topics pertinent to almost any field. The close association between the process of algorithm discovery and that of general problem solving has caused computer scientists to join with those of other disciplines in the search for better problem-solving techniques. Ultimately, one would like to reduce the process of problem solving to an algorithm in itself, but this has been shown to be impossible. (This is a result of the material in Chapter 12, where we will show that there are problems that do not have algorithmic solutions.) Thus the ability to solve problems remains more of an artistic skill to be developed than a precise science to be learned.

As evidence of the elusive, artistic nature of problem solving, the following loosely defined problem-solving phases presented by the mathematician G. Polya in 1945 remain the basic principles on which many attempts to teach problem-solving skills are based today.

Phase 1. Understand the problem.

Phase 2. Devise a plan for solving the problem.

Phase 3. Carry out the plan.

Phase 4. Evaluate the solution for accuracy and for its potential as a tool for solving other problems.

Translated into the context of program development, these phases become

Phase 1. Understand the problem.

Phase 2. Get an idea of how an algorithmic function might solve the problem.

Phase 3. Formulate the algorithm and represent it as a program.

Phase 4. Evaluate the program for accuracy and for its potential as a tool for solving other problems.

Having presented Polya's list, we should emphasize that these phases are not steps to be followed when trying to solve a problem but rather phases that will be completed sometime during the solution process. The key word here is *followed.* You do not solve problems by following. Rather, to solve a problem, you must take the initiative and lead. If you approach the task of solving a problem in the frame of mind depicted by "Now I've finished Phase 1, it's time to move on to Phase 2," you are not likely to be successful. However, if you become involved with the problem and ultimately solve it, you most likely can look back at what you did and realize that you performed Polya's phases.

Another important observation is that Polya's phases are not necessarily completed in sequence. Successful problem solvers often start formulating strategies for solving a problem (Phase 2) before the problem itself is entirely understood (Phase 1). Then, if these strategies fail (during Phases 3 or 4), the potential problem solver gains a deeper understanding of the intricacies of the problem and, with this deeper understanding, can return to form other and hopefully more successful strategies.

Keep in mind that we are discussing how problems are solved—not how we would like them to be solved. Ideally, we would like to eliminate the waste inherent in the trial-and-error process just described. In the case of developing large software systems, discovering a misunderstanding as late as Phase 4 can represent a tremendous loss in resources. Avoiding such catastrophes is a major goal of software engineers (Chapter 7), who have traditionally insisted on a thorough understanding of a problem before proceeding with a solution. One could argue, however, that a true understanding of a problem is not obtained until a solution has been found. The mere fact that a problem is unsolved implies a lack of understanding. To insist on a complete understanding of the problem before proposing any solutions is therefore somewhat idealistic.

As an example, consider the following problem:

> Person A is charged with the task of determining the ages of person B's three children. B tells A that the product of the children's ages is 36. After considering this clue, A replies that another clue is required, so B tells A the sum of the children's ages. Again, A replies that another clue is needed, so B tells A that the oldest child plays the piano. After hearing this clue, A tells B the ages of the three children.
>
> How old are the three children?

At first glance the last clue seems to be totally unrelated to the problem, yet it is apparently this clue that allows A to finally determine the ages of the children. How can this be? Let us proceed by formulating a plan of attack and following this plan, even though we still have many questions about the problem. Our plan will be to trace the steps described by the problem statement while keeping track of the information available to person A as the story progresses.

The first clue given A is that the product of the children's ages is 36. This means that the triple representing the three ages is one of those listed in Figure 5.5(a). The next clue is the sum of the desired triple. We are not told what this sum is, but we are told that this information is not enough for A to isolate the correct triple; therefore the desired triple must be one whose sum appears at least twice in the table of Figure 5.5(b). But the only triples appearing in Figure 5.5(b) with identical sums are (1,6,6) and (2,2,9), both of which produce the sum 13. This is the information available to A at the time the last clue is given. It is at this point that we finally understand the significance of the last clue. It has nothing to do with playing the piano; rather it is the fact that there is an oldest child. This rules out the triple (1,6,6) and thus allows us to conclude that the children's ages are 2, 2, and 9.

In this case, then, it is not until we attempt to implement our plan for solving the problem (Phase 3) that we gain a complete understanding of the problem (Phase 1). Had we insisted on completing Phase 1 before proceeding, we would probably never have found the children's ages. Such irregularities in the problem-solving process are fundamental to the difficulties in developing systematic approaches to problem solving.

Figure 5.5 Analyzing the possibilities

a. Triples whose product is 36

(1,1,36)	(1,6,6)
(1,2,18)	(2,2,9)
(1,3,12)	(2,3,6)
(1,4,9)	(3,3,4)

b. Sums of triples from part (a)

1 + 1 + 36 = 38	1 + 6 + 6 = 13
1 + 2 + 18 = 21	2 + 2 + 9 = 13
1 + 3 + 12 = 16	2 + 3 + 6 = 11
1 + 4 + 9 = 14	3 + 3 + 4 = 10

Another irregularity is the mysterious inspiration that might come to a potential problem solver who, having worked on a problem without apparent success, at a later time suddenly sees the solution while doing another task. This phenomenon was identified by H. von Helmholtz as early as 1896 and was discussed by the mathematician Henri Poincaré in a lecture before the Psychological Society in Paris. There, Poincaré described his experiences of realizing the solution to a problem he had worked on after he had set it aside and begun other projects. The phenomenon reflects a process in which a subconscious part of the mind appears to continue working and, if successful, forces the solution into the conscious mind. Today, the period between conscious work on a problem and the sudden inspiration is known as an incubation period, and its understanding remains a goal of current research.

Getting a Foot in the Door

We have been discussing problem solving from a somewhat philosophical point of view while avoiding a direct confrontation with the question of how we should go about trying to solve a problem. There are, of course, numerous problem-solving approaches, each of which can be successful in certain settings. We will identify some of them shortly. For now, we note that there seems to be a common thread running through these techniques, which simply stated is "get your foot in the door." As an example, let us consider the following simple problem:

> Before A, B, C, and D ran a race they made the following predictions:
>
> A predicted that B would win.
> B predicted that D would be last.
> C predicted that A would be third.
> D predicted that A's prediction would be correct.
>
> Only one of these predictions was true, and this was the prediction made by the winner. In what order did A, B, C, and D finish the race?

After reading the problem and analyzing the data, it should not take long to realize that since the predictions of A and D were equivalent and only one prediction was true, the predictions of both A and D must be false. Thus neither A nor D were winners. At this point we have our foot in the door, and obtaining the complete solution to our problem is merely a matter of extending our knowledge from here. If A's prediction was false, then B did not win either. The only remaining choice for the winner is C. Thus, C won the race, and C's prediction was true. Consequently, we know that A came in third. That means that the finishing order was either CBAD or CDAB. But the former is ruled out because B's prediction must be false. Therefore the finishing order was CDAB.

Of course, being told to get our foot in the door is not the same as being told how to do it. Obtaining this toehold, as well as realizing how to expand this initial thrust into a complete solution to the problem, requires creative input from the would-be problem solver. There are, however, several general approaches that have been proposed by Polya and others for how one might go about getting a foot in the door. One is to try working the problem backward. For instance, if the problem is to find a way of producing a particular output from a given input, one might start with that output and attempt to back up to the given input. This approach is typical of people trying to discover the bird-folding algorithm in the previous section. They tend to unfold a completed bird in an attempt to see how it is constructed.

Another general problem-solving approach is to look for a related problem that is either easier to solve or has been solved before and then try to apply its solution to the current problem. This technique is of particular value in the context of program development. Generally, program development is not the process of solving a particular instance of a problem but rather of finding a general algorithm that can be used to solve all instances of the problem. More precisely, if we were faced with the task of developing a program for alphabetizing lists of names, our task would not be to sort a particular list but to find a general algorithm that could be used to sort any list of names. Thus, although the instructions

```
Interchange the names David and Alice.
Move the name Carol to the position between Alice and David.
Move the name Bob to the position between Alice and Carol.
```

correctly sort the list David, Alice, Carol, and Bob, they do not constitute the general-purpose algorithm we desire. What we need is an algorithm that can sort this list as well as other lists we might encounter. This is not to say that our solution for sorting a particular list is totally worthless in our search for a general-purpose algorithm. We might, for instance, get our foot in the door by considering such special cases in an attempt to find general principles that can in turn be used to develop the desired general-purpose algorithm. In this case, then, our solution is obtained by the technique of solving a collection of related problems.

Still another approach to getting a foot in the door is to apply **stepwise refinement,** which is essentially the technique of not trying to conquer an entire task (in all its detail) at once. Rather, stepwise refinement proposes that one first view the problem at hand in terms of several subproblems. The idea is that by breaking the original problem into subproblems, one is able to approach the overall solution in terms of steps, each of which is easier to solve than the entire original problem. In turn, stepwise refinement proposes that these steps be decomposed into smaller steps and these smaller steps be broken into still smaller ones until the entire problem has been reduced to a collection of easily solved subproblems.

In this light, stepwise refinement is a **top-down methodology** in that it progresses from the general to the specific. In contrast, a **bottom-up methodology** progresses from the specific to the general. Although contrasting in theory, the two approaches often complement each other in creative problem solving. The decomposition of a problem proposed by the top-down methodology of stepwise refinement is often guided by the problem solver's intuition, which might be working in a bottom-up mode.

The top-down methodology of stepwise refinement is essentially an organizational tool whose problem-solving attributes are consequences of this organization. It has long been an important design methodology in the data processing community, where the development of large software systems encompasses a significant organizational component. But, as we will learn in Chapter 7, large software systems are increasingly being constructed by combining prefabricated components—an approach that is inherently bottom-up. Thus, both top-down and bottom-up methodologies remain important tools in computer science.

The importance of maintaining such a broad perspective is exemplified by the fact that bringing preconceived notions and preselected tools to the problem-solving task can sometimes mask a problem's simplicity. The ages-of-the-children problem discussed earlier in this section is an excellent example of this phenomenon. Students of algebra invariably approach the problem as a system of simultaneous equations, an approach that leads to a dead end and often traps the would-be problem solver into believing that the information given is not sufficient to solve the problem.

Another example is the following:

> As you step from a pier into a boat, your hat falls into the water, unbeknownst to you. The river is flowing at 2.5 miles per hour so your hat begins to float downstream. In the meantime, you begin traveling upstream in the boat at a speed of 4.75 miles per hour relative to the water. After 10 minutes you realize that your hat is missing, turn the boat around, and begin to chase your hat down the river. How long will it take to catch up with your hat?

Most algebra students as well as calculator enthusiasts approach this problem by first determining how far upstream the boat will have traveled in 10 minutes as well as how far downstream the hat will have traveled during that same time. Then, they determine how long it will take for the boat to travel downstream to this position. But, when the boat reaches this position, the hat will have floated farther downstream. Thus, the problem solver either begins to apply techniques of calculus or becomes trapped in a cycle of computing where the hat will be each time the boat goes to where the hat was.

The problem is much simpler than this, however. The trick is to resist the urge to begin writing formulas and making calculations. Instead, we need to put these skills aside and adjust our perspective. The entire problem takes place in the river. The fact that the water is moving in relation to the shore is irrelevant. Think of the same problem posed on a large conveyor belt instead of a river. First, solve the problem with the conveyor belt at rest. If you place your hat at your feet while standing on the belt and then walk away from your hat for 10 minutes, it will take 10 minutes to return to your hat. Now turn on the conveyor belt. This means that the scenery will begin to move past the belt, but, because you are on the belt, this does not change your relationship to the belt or your hat. It will still take 10 minutes to return to your hat.

We conclude that algorithm discovery remains a challenging art that must be developed over a period of time rather than taught as a subject consisting of well-defined methodologies. Indeed, to train a potential problem solver to follow certain methodologies is to quash those creative skills that should instead be nurtured.

Questions & Exercises

1. **a.** Find an algorithm for solving the following problem: Given a positive integer n, find the list of positive integers whose product is the largest among all the lists of positive integers whose sum is n. For example, if n is 4, the desired list is 2, 2 because 2×2 is larger than $1 \times 1 \times 1 \times 1$, $2 \times 1 \times 1$, and 3×1. If n is 5, the desired list is 2, 3.

   b. What is the desired list if $n = 2001$?

   c. Explain how you got your foot in the door.

2. a. Suppose we are given a checkerboard consisting of 2^n rows and 2^n columns of squares, for some positive integer n, and a box of L-shaped tiles, each of which can cover exactly three squares on the board. If any single square is cut out of the board, can we cover the remaining board with tiles such that tiles do not overlap or hang off the edge of the board?
   b. Explain how your solution to (a) can be used to show that $2^{2n} - 1$ is divisible by 3 for all positive integers n.
   c. How are (a) and (b) related to Polya's phases of problem solving?
3. Decode the following message, then explain how you got your foot in the door. *Pdeo eo pda yknnayp wjosan.*
4. Would you be following a top-down methodology if you attempted to solve a picture puzzle merely by pouring the pieces out on a table and trying to piece them together? Would your answer change if you looked at the puzzle box to see what the entire picture was supposed to look like?

5.4 Iterative Structures

Our goal now is to study some of the repetitive structures used in describing algorithmic processes. In this section we discuss **iterative structures** in which a collection of instructions is repeated in a looping manner. In the next section we will introduce the technique of recursion. As a side effect, we will introduce some popular algorithms—the sequential search, the binary search, and the insertion sort. We begin by introducing the sequential search algorithm.

The Sequential Search Algorithm

Consider the problem of searching within a list for the occurrence of a particular target value. We want to develop an algorithm that determines whether that value is in the list. If the value is in the list, we consider the search a success; otherwise we consider it a failure. We assume that the list is sorted according to some rule for ordering its entries. For example, if the list is a list of names, we assume the names appear in alphabetical order, or if the list consists of numeric values, we assume its entries appear in order of increasing magnitude.

To get our foot in the door, we imagine how we might search a guest list of perhaps 20 entries for a particular name. In this setting we might scan the list from its beginning, comparing each entry with the target name. If we find the target name, the search terminates as a success. However, if we reach the end of the list without finding the target value, our search terminates as a failure. In fact, if we reach a name greater than (alphabetically) the target name without finding the target, our search terminates as a failure. (Remember, the list is arranged in alphabetical order, so reaching a name greater than the target name indicates that the target does not appear in the list.) In summary, our rough idea is to continue searching down the list as long as there are more names to be investigated and the target name is greater than the name currently being considered.

In our pseudocode, this process can be represented as

```
Select the first entry in the list as TestEntry.
while (TargetValue > TestEntry and entries remain):
  Select the next entry in the list as TestEntry.
```

Upon terminating this `while` structure, one of two conditions will be true: Either the target value has been found or the target value is not in the list. In either case we can detect a successful search by comparing the test entry to the target value. If they are equal, the search has been successful. Thus we add the statement

```
if (TargetValue == TestEntry):
  Declare the search a success.
else:
  Declare the search a failure.
```

to the end of our pseudocode routine.

Finally, we observe that the first statement in our routine, which selects the first entry in the list as the test entry, is based on the assumption that the list in question contains at least one entry. We might reason that this is a safe guess, but just to be sure, we can position our routine as the else option of the statement

```
if (List is empty):
  Declare search a failure.
else:
  . . .
```

This produces the function shown in Figure 5.6. Note that this function can be used from within other functions by using statements such as

```
Search() the passenger list using Darrel Baker as the target value.
```

to find out if Darrel Baker is a passenger and

```
Search() the list of ingredients using nutmeg as the target value.
```

to find out if nutmeg appears in the list of ingredients.

Figure 5.6 The sequential search algorithm in pseudocode

```
def Search(List, TargetValue):
  if (List is empty):
    Declare search a failure.
  else:
    Select the first entry in List to be TestEntry.
    while (TargetValue > TestEntry and
        there remain entries to be considered):
      Select the next entry in List as TestEntry.
    if (TargetValue == TestEntry):
      Declare search a success.
    else:
      Declare search a failure.
```

In summary, the algorithm represented by Figure 5.6 considers the entries in the sequential order in which they occur in the list. For this reason, the algorithm is called the **sequential search** algorithm. Because of its simplicity, it is often used for short lists or when other concerns dictate its use. However, in the case of long lists, sequential searches are not as efficient as other techniques (as we shall soon see).

Loop Control

The repetitive use of an instruction or sequence of instructions is an important algorithmic concept. One method of implementing such repetition is the iterative structure known as the **loop,** in which a collection of instructions, called the **body** of the loop, is executed in a repetitive fashion under the direction of some control process. A typical example is found in the sequential search algorithm represented in Figure 5.6. Here we use a `while` statement to control the repetition of the single statement `Select the next entry in List as the TestEntry`. Indeed, the `while` statement

```
while (condition):
  Body
```

exemplifies the concept of a loop structure in that its execution traces the cyclic pattern

```
check the condition.
execute the body.
check the condition.
execute the body.
  .
  .
  .
check the condition.
```

until the condition fails.

As a general rule, the use of a loop structure produces a higher degree of flexibility than would be obtained merely by explicitly writing the body several times. For example, to execute the statement

```
Add a drop of sulfuric acid.
```

three times, we could write:

```
Add a drop of sulfuric acid.
Add a drop of sulfuric acid.
Add a drop of sulfuric acid.
```

But we cannot produce a similar sequence that is equivalent to the loop structure

```
while (the pH level is greater than 4):
  add a drop of sulfuric acid
```

because we do not know in advance how many drops of acid will be required.

Let us now take a closer look at the composition of loop control. You might be tempted to view this part of a loop structure as having minor importance. After all, it is typically the body of the loop that actually performs the task at hand (for example, adding drops of acid)—the control activities appear merely as the overhead involved because we chose to execute the body in a repetitive fashion. However, experience has shown that the control of a loop is the more error-prone part of the structure and therefore deserves our attention.

The control of a loop consists of the three activities initialize, test, and modify (Figure 5.7), with the presence of each being required for successful loop control. The test activity has the obligation of causing the termination of the looping process by watching for a condition that indicates termination should take place. This condition is known as the **termination condition.** It is for the purpose of this test activity that we provide a condition within each `while` statement of our pseudocode. In the case of the `while` statement, however, the condition stated is the condition under which the body of the loop should be executed—the termination condition is the negation of the condition appearing in the `while` structure. Thus, in the statement

```
while (the pH level is greater than 4):
  add a drop of sulfuric acid
```

the termination condition is "the pH level is *not* greater than 4," and in the `while` statement of Figure 5.6, the termination condition could be stated as

```
(TargetValue <= TestEntry) or (there are no more entries to be considered)
```

The other two activities in the loop control ensure that the termination condition will ultimately occur. The initialization step establishes a starting condition, and the modification step moves this condition toward the termination condition. For instance, in Figure 5.6, initialization takes place in the statement preceding the `while` statement, where the current test entry is established as the first list entry. The modification step in this case is actually accomplished within the loop body, where our position of interest (identified by the test entry) is moved toward the end of the list. Thus, having executed the initialization step, repeated application of the modification step results in the termination condition being reached. (Either we will reach a test entry that is greater than or equal to the target value or we ultimately reach the end of the list.)

We should emphasize that the initialization and modification steps must lead to the appropriate termination condition. This characteristic is critical for proper loop control, and thus one should always double-check for its presence when designing a loop structure. Failure to make such an evaluation can

Figure 5.7 Components of repetitive control

Initialize:	Establish an initial state that will be modified toward the termination condition
Test:	Compare the current state to the termination condition and terminate the repetition if equal
Modify:	Change the state in such a way that it moves toward the termination condition

lead to errors even in the simplest cases. A typical example is found in the statements

```
Number = 1
while (Number != 6):
  Number = Number + 2
```

Note the use of the Python "`!=`" operator, which we read as, "not equal". Here the termination condition is "`Number == 6`." But the value of `Number` is initialized at 1 and then incremented by 2 in the modification step. Thus, as the loop cycles, the values assigned to `Number` will be 1, 3, 5, 7, 9, and so on, but never the value 6. As a result, this loop will never terminate.

The order in which the components of loop control are executed can have subtle consequences. In fact, there are two common loop structures that differ merely in this regard. The first is exemplified by our pseudocode statement

```
while (condition):
  Activity
```

whose semantics are represented in Figure 5.8 in the form of a **flowchart.** (Such charts use various shapes to represent individual steps and use arrows to indicate the order of the steps. The distinction between the shapes indicates the type of action involved in the associated step. A diamond indicates a decision and a rectangle indicates an arbitrary statement or sequence of statements.) Note that the test for termination in the while structure occurs before the loop's body is executed.

In contrast, the structure in Figure 5.9 requests that the body of the loop be executed before the test for termination is performed. In this case, the loop's body is always performed at least once, whereas in the `while` structure, the body is never executed if the termination condition is satisfied the first time it is tested.

Python does not have a built-in structure for this second kind of loop, although it is easy enough to build an equivalent using the existing `while` structure with an `if`-statement and a `break` at the end of the loop body. For our pseudocode,

Figure 5.8 The while loop structure

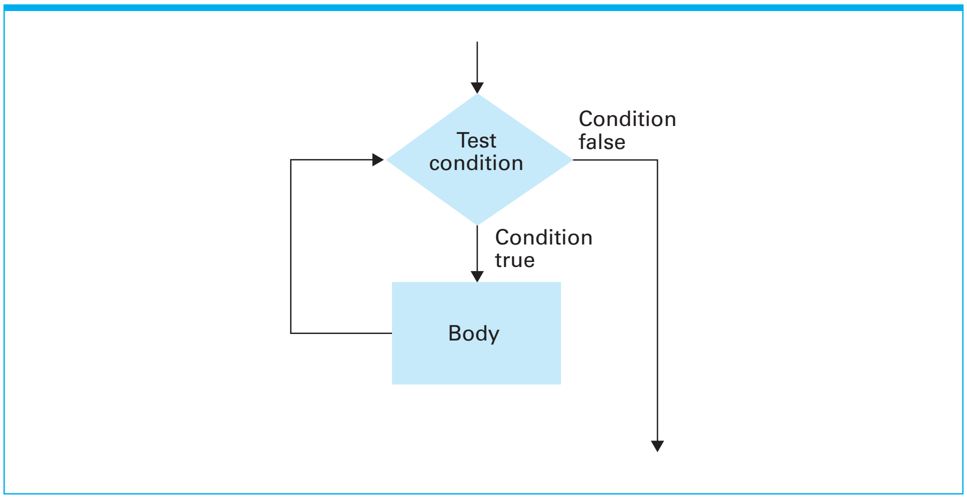
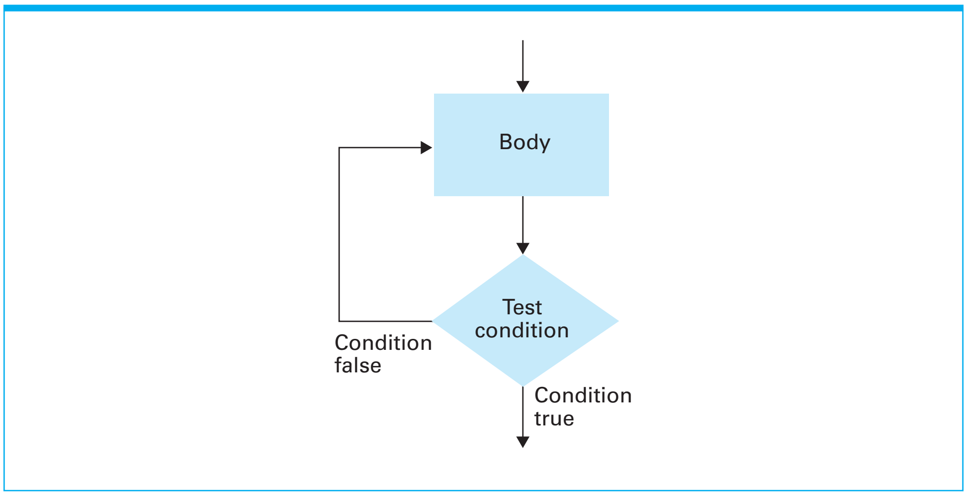


Figure 5.9 The repeat loop structure



we will borrow keywords that exists in several other languages, using the syntactic form

```
repeat:
  activity
  until (condition)
```

to represent the structure shown in Figure 5.9. Thus, the statement

```
repeat:
  take a coin from your pocket
  until (there are no coins in your pocket)
```

assumes there is a coin in your pocket at the beginning, but

```
while (there is a coin in your pocket):
  take a coin from your pocket
```

does not.

Following the terminology of our pseudocode, we will usually refer to these structures as the `while` loop structure or the `repeat` loop structure. In a more generic context you might hear the `while` loop structure referred to as a **pretest loop** (since the test for termination is performed before the body is executed) and the `repeat` loop structure referred to as a **posttest loop** (since the test for termination is performed after the body is executed).

While many algorithms require careful consideration of the initialization, test, and modify activities when controlling loops, others follow some very common patterns. Particularly when working with lists of data, the most common pattern is to start with the first element in a list and consider each element in the list until the end is reached. Returning briefly to our sequential search example, we saw the pattern similar to

```
Select the first entry in the list
while (there remain entries to be considered):
  . . .
  Select the next entry in the list
```

Because this structure occurs so frequently in algorithms, we will use the syntactic form

```
for Item in List:
  . . .
```

to describe a loop that iterates through each element of a list. Notice that this pseudocode primitive is effectively one level of abstraction higher than the `while` structure, because we can accomplish the same effect with separate initialize, modify, and test structures, but this version more succinctly conveys the meaning of the loop without unnecessary detail.

Each time through the body of this `for` loop structure, the value `Item` will become the next element in `List`. The termination condition of this loop is implicitly when the end of `List` is reached. As an example, to total up the numbers in a list we could use

```
Sum = 0
for Number in List:
  Sum = Sum + Number
```

This type of loop is often called a **for-each** loop in languages other than Python and is a special case of the pretest loop. The `for` structure is best suited to situations in which the algorithm will perform the same steps on each element in a list, and it is not necessary to separately keep track of a loop counting variable.

The Insertion Sort Algorithm

As an additional example of using iterative structures, let us consider the problem of sorting a list of names into alphabetical order. But before proceeding, we should identify the constraints under which we will work. Simply stated, our goal is to sort the list "within itself." In other words, we want to sort the list by shuffling its entries as opposed to moving the list to another location. Our situation is analogous to the problem of sorting a list whose entries are recorded on separate index cards spread out on a crowded desktop. We have cleared off enough space for the cards but are not allowed to push additional materials back to make more room. This restriction is typical in computer applications, not because the workspace within the machine is necessarily crowded like our desktop, but simply because we want to use the storage space available in an efficient manner.

Let us get a foot in the door by considering how we might sort the names on the desktop. Consider the list of names

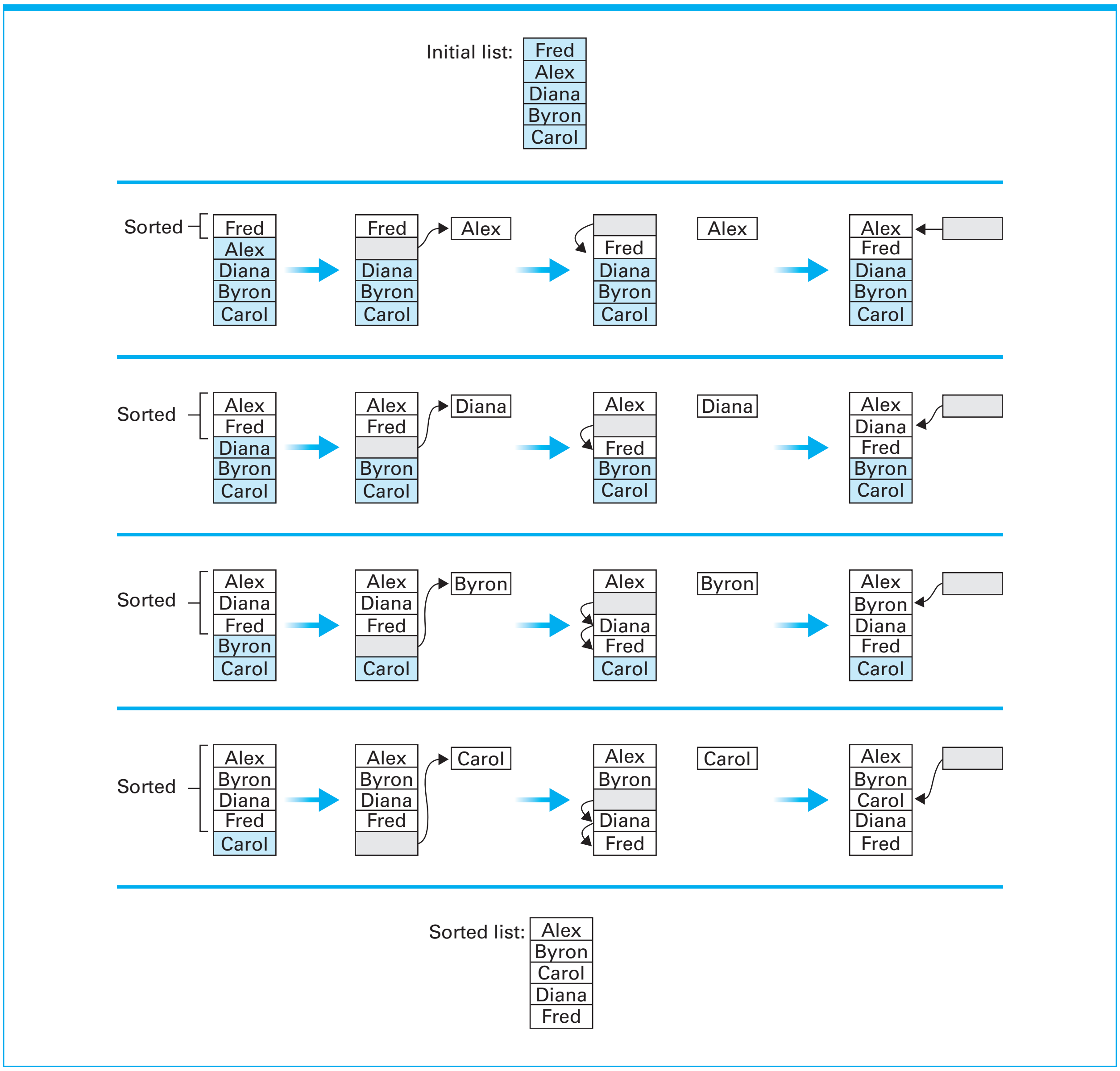
Fred
Alex
Diana
Byron
Carol

One approach to sorting this list is to note that the sublist consisting of only the top name, Fred, is sorted but the sublist consisting of the top two names, Fred and Alex, is not. Thus we might pick up the card containing the name Alex, slide the name Fred down into the space where Alex was, and then place the name Alex in the hole at the top of the list, as represented by the first row in Figure 5.10. At this point our list would be

Alex
Fred
Diana
Byron
Carol

Now the top two names form a sorted sublist, but the top three do not. Thus we might pick up the third name, Diana, slide the name Fred down into the hole

Figure 5.10 Sorting the list Fred, Alex, Diana, Byron, and Carol alphabetically



where Diana was, and then insert Diana in the hole left by Fred, as summarized in the second row of Figure 5.10. The top three entries in the list would now be sorted. Continuing in this fashion, we could obtain a list in which the top four entries are sorted by picking up the fourth name, Byron, sliding the names Fred and Diana down, and then inserting Byron in the hole (see the third row of Figure 5.10). Finally, we can complete the sorting process by picking up Carol, sliding Fred and Diana down, and then inserting Carol in the remaining hole (see the fourth row of Figure 5.10).

Having analyzed the process of sorting a particular list, our task now is to generalize this process to obtain an algorithm for sorting general lists. To this end, we observe that each row of Figure 5.10 represents the same general process: Pick up the first name in the unsorted portion of the list, slide the names greater than the extracted name down, and insert the extracted name back in the list where the hole appears. If we identify the extracted name as the pivot entry, this process can be expressed in our pseudocode as

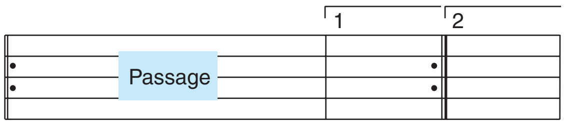
```
Move the pivot entry to a temporary location leaving a hole in List
while (there is a name above the hole and
  that name is greater than the pivot):
  move the name above the hole down into the hole
    leaving a hole above the name
Move the pivot entry into the hole in List.
```

Iterative Structures in Music

Musicians were using and programming iterative structures centuries before computer scientists. Indeed, the structure of a song (being composed of multiple verses, each followed by the chorus) is exemplified by the `while` statement

```
while (there is a verse remaining):
  sing the next verse
  sing the chorus
```

Moreover, the notation



is merely a composer's way of expressing the structure

```
N = 1
while (N < 3):
  play the passage
  play the Nth ending
  N = N + 1
```

Next, we observe that this process should be executed repeatedly. To begin the sorting process, the pivot should be the second entry in the list and then, before each additional execution, the pivot selection should be one more entry down the list until the last entry has been positioned. That is, as the preceding routine is repeated, the initial position of the pivot entry should advance from the second entry to the third, then to the fourth, etc., until the routine has positioned the last entry in the list. Following this lead we can control the required repetition with the statements

```
N = 2
while (the value of N does not exceed the length of List):
  Select the Nth entry in List as the pivot entry
    .
    .
    .
  N = N + 1
```

where `N` represents the position to use for the pivot entry, `the length of List` refers to the number of entries in the list, and the dots indicate the location where the previous routine should be placed.

Our complete pseudocode program is shown in Figure 5.11. In short, the program sorts a list by repeatedly removing an entry and inserting it into its proper place. It is because of this repeated insertion process that the underlying algorithm is called the **insertion sort.**

Note that the structure of Figure 5.11 is that of a loop within a loop, the outer loop being expressed by the first `while` statement and the inner loop represented by the second `while` statement. Each execution of the body of the outer loop results in the inner loop being initialized and executed until its termination condition is obtained. Thus, a single execution of the outer loop's body will result in several executions of the inner loop's body.

The initialization component of the outer loop's control consists of establishing the initial value of `N` with the statement

```
N = 2
```

Figure 5.11 The insertion sort algorithm expressed in pseudocode

```
def Sort (List):
  N = 2
  while (the value of N does not exceed the length of List):
    Select the Nth entry in List as the pivot entry.
    Move the pivot entry to a temporary location leaving
      a hole in List.
    while (there is a name above the hole and that name
        is greater than the pivot):
      Move the name above the hole down into the hole
        leaving a hole above the name.
    Move the pivot entry into the hole in List.
    N = N + 1
```

The modification component is handled by incrementing the value of N at the end of the loop's body with the statement

```
N = N + 1
```

The termination condition occurs when the value of N exceeds the length of the list.

The inner loop's control is initialized by removing the pivot entry from the list and thus creating a hole. The loop's modification step is accomplished by moving entries down into the hole, thus causing the hole to move up. The termination condition consists of the hole being immediately below a name that is not greater than the pivot or of the hole reaching the top of the list.

Questions & Exercises

1. Modify the sequential search function in Figure 5.6 to allow for lists that are not sorted.
2. Convert the pseudocode routine

```
Z = 0
X = 1
while (X < 6):
  Z = Z + X
  X = X + 1
```

   to an equivalent routine using a `repeat` statement.
3. Some of the popular programming languages today use the syntax

```
while (. . . ) do (. . . )
```

   to represent a pretest loop and the syntax

```
do (. . . ) while (. . . )
```

   to represent a posttest loop. Although elegant in design, what problems could result from such similarities?
4. Suppose the insertion sort as presented in Figure 5.11 was applied to the list Gene, Cheryl, Alice, and Brenda. Describe the organization of the list at the end of each execution of the body of the outer `while` structure.
5. Why would we not want to change the phrase "greater than" in the `while` statement in Figure 5.11 to "greater than or equal to"?
6. A variation of the insertion sort algorithm is the **selection sort.** It begins by selecting the smallest entry in the list and moving it to the front. It then selects the smallest entry from the remaining entries in the list and moves it to the second position in the list. By repeatedly selecting the smallest entry from the remaining portion of the list and moving that entry forward, the sorted version of the list grows from the front of the list, while the back portion of the list consisting of the remaining unsorted entries shrinks. Use our pseudocode to express a function similar to that in Figure 5.11 for sorting a list using the selection sort algorithm.

7. Another well-known sorting algorithm is the **bubble sort.** It is based on the process of repeatedly comparing two adjacent names and interchanging them if they are not in the correct order relative to each other. Let us suppose that the list in question has *n* entries. The bubble sort would begin by comparing (and possibly interchanging) the entries in positions *n* and *n – 1.* Then, it would consider the entries in positions *n – 1* and *n – 2,* and continue moving forward in the list until the first and second entries in the list had been compared (and possibly interchanged). Observe that this pass through the list will pull the smallest entry to the front of the list. Likewise, another such pass will ensure that the next to the smallest entry will be pulled to the second position in the list. Thus, by making a total of *n – 1* passes through the list, the entire list will be sorted. (If one watches the algorithm at work, one sees the small entries bubble to the top of the list—an observation from which the algorithm gets its name.) Use our pseudocode to express a function similar to that in Figure 5.11 for sorting a list using the bubble sort algorithm.

5.5 Recursive Structures

Recursive structures provide an alternative to the loop paradigm for implementing the repetition of activities. Whereas a loop involves repeating a set of instructions in a manner in which the set is completed and then repeated, recursion involves repeating the set of instructions as a subtask of itself. As an analogy, consider the process of conducting telephone conversations with the call waiting feature. There, an incomplete telephone conversation is set aside while another incoming call is processed. The result is that two conversations take place. However, they are not performed one-after-the-other as in a loop structure, but instead one is performed within the other.

The Binary Search Algorithm

As a way of introducing recursion, let us again tackle the problem of searching to see whether a particular entry is in a sorted list, but this time we get our foot in the door by considering the procedure we follow when searching a dictionary. In this case we do not perform a sequential entry-by-entry or even a page-by-page procedure. Rather, we begin by opening the directory to a page in the area where we believe the target entry is located. If we are lucky, we will find the target value there; otherwise, we must continue searching. But at this point we will have narrowed our search considerably.

Of course, in the case of searching a dictionary, we have prior knowledge of where words are likely to be found. If we are looking for the word *somnambulism,* we would start by opening to the latter portion of the dictionary. In the case of generic lists, however, we do not have this advantage, so let us agree to always start our search with the "middle" entry in the list. Here we write the word *middle* in quotation marks because the list might have an even number of entries and thus no middle entry in the exact sense. In this case, let us agree that the "middle" entry refers to the first entry in the second half of the list.

If the middle entry in the list is the target value, we can declare the search a success. Otherwise, we can at least restrict the search process to the first or last half of the list depending on whether the target value is less than or greater than the entry we have considered. (Remember that the list is sorted.)

To search the remaining portion of the list, we could apply the sequential search, but instead let us apply the same approach to this portion of the list that we used for the whole list. That is, we select the middle entry in the remaining portion of the list as the next entry to consider. As before, if that entry is the target value, we are finished. Otherwise we can restrict our search to an even smaller portion of the list.

This approach to the searching process is summarized in Figure 5.12, where we consider the task of searching the list on the left of the figure for the entry John. We first consider the middle entry Harry. Since our target belongs after this entry, the search continues by considering the lower half of the original list. The middle of this sublist is found to be Larry. Since our target should precede Larry, we turn our attention to the first half of the current sublist. When we interrogate the middle of that secondary sublist, we find our target John and declare the search a success. In short, our strategy is to successively divide the list in question into smaller segments until the target is found or the search is narrowed to an empty segment.

We need to emphasize this last point. If the target value is not in the original list, our approach to searching the list will proceed by dividing the list into smaller segments until the segment under consideration is empty. At this point our algorithm should recognize that the search is a failure.

Figure 5.13 is a first draft of our thoughts using our pseudocode. It directs us to begin a search by testing to see if the list is empty. If so, we are told to report that the search is a failure. Otherwise, we are told to consider the middle entry in the list. If this entry is not the target value, we are told to search either the front half or the back half of the list. Both of these possibilities require a secondary search. It would be nice to perform these searches by calling on the services of

Figure 5.12 Applying our strategy to search a list for the entry John

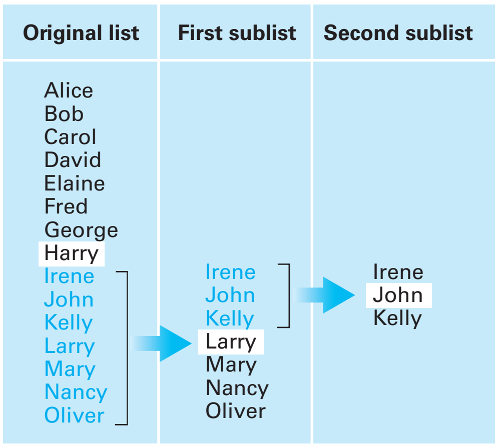


Figure 5.13 A first draft of the binary search technique

```
if (List is empty):
  Report that the search failed.
else:
  TestEntry = the "middle" entry in the List
  if (TargetValue == TestEntry):
    Report that the search succeeded.
  if (TargetValue < TestEntry):
    Search() the portion of List preceding TestEntry for TargetValue,
      and report the result of that search.
  if (TargetValue > TestEntry):
    Search() the portion of List following TestEntry for TargetValue,
      and report the result of that search.
```

an abstract tool. In particular, our approach is to apply a function named Search to carry out these secondary searches. To complete our program, therefore, we must provide such a function.

But this function should perform the same task that is expressed by the pseudocode we have already written. It should first check to see if the list it is given is empty, and if it is not, it should proceed by considering the middle entry of that list. Thus we can supply the function we need merely by identifying the current routine as being the function named `Search` and inserting references to that function where the secondary searches are required. The result is shown in Figure 5.14.

Note that this function contains a reference to itself. If we were following this function and came to the instruction

```
Search(. . . )
```

we would apply the same function to the smaller list that we were applying to the original one. If that search succeeded, we would return to declare our original search successful; if this secondary search failed, we would declare our original search a failure.

Searching and Sorting

The sequential and binary search algorithms are only two of many algorithms for performing the search process. Likewise, the insertion sort is only one of many sorting algorithms. Other classic algorithms for sorting include the merge sort (discussed in Chapter 12), the selection sort (question 6 in Section 5.4), the bubble sort (question 7 in Section 5.4), the quick sort (which applies a divide-and-conquer approach to the sorting process), and the heap sort (which uses a clever technique for finding the entries that should be moved forward in the list). You will find discussions of these algorithms in the books listed under Additional Reading at the end of this chapter.

Figure 5.14 The binary search algorithm in pseudocode

```
def Search(List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

To see how the function in Figure 5.14 performs its task, let us follow it as it searches the list Alice, Bill, Carol, David, Evelyn, Fred, and George, for the target value Bill. Our search begins by selecting David (the middle entry) as the test entry under consideration. Since the target value (Bill) is less than this test entry, we are instructed to apply the function `Search` to the list of entries preceding David—that is, the list Alice, Bill, and Carol. In so doing, we create a second copy of the search function and assign it to this secondary task.

We now have two copies of our search function being executed, as summarized in Figure 5.15. Progress in the original copy is temporarily suspended at the instruction

```
Search(Sublist, TargetValue)
```

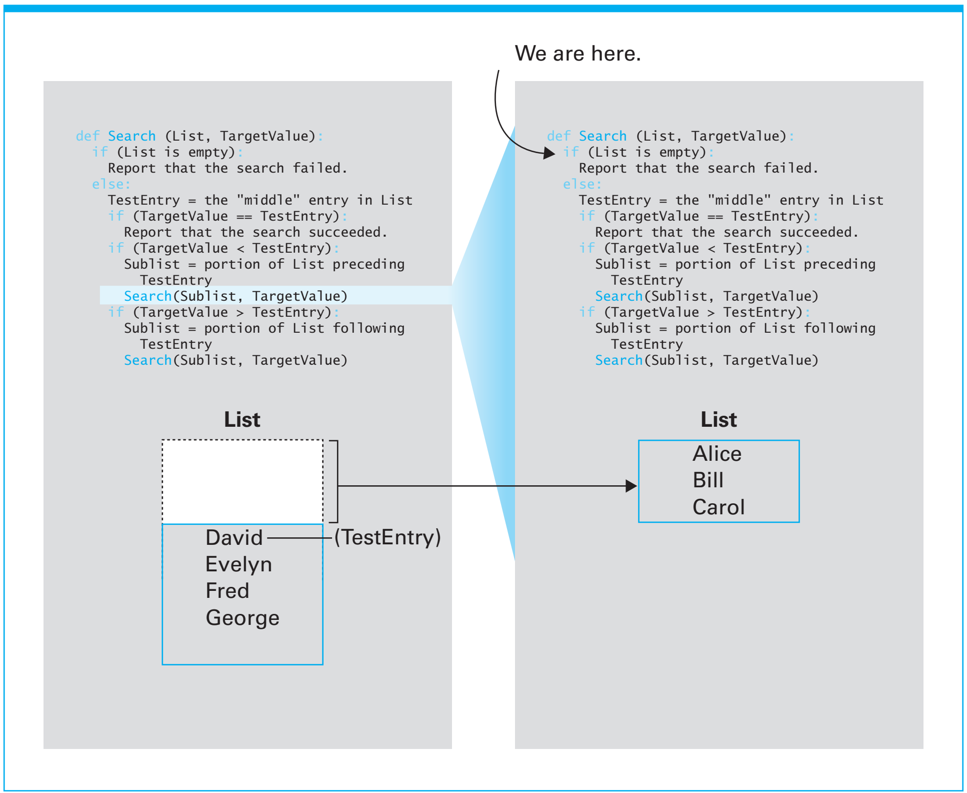
while we apply the second copy to the task of searching the list Alice, Bill, and Carol. When we complete this secondary search, we will discard the second copy of the function, report its findings to the original copy, and continue progress in the original. In this way, the second copy of the function executes as a subordinate to the original, performing the task requested by the original module and then disappearing.

The secondary search selects Bill as its test entry because that is the middle entry in the list Alice, Bill, and Carol. Since this is the same as the target value, it declares its search to be a success and terminates.

At this point, we have completed the secondary search as requested by the original copy of the function, so we are able to continue the execution of that original copy. Here we are told that the result of the secondary search should be reported as the result of the original search. Thus we report that the original search has succeeded. Our process has correctly determined that Bill is a member of the list Alice, Bill, Carol, David, Evelyn, Fred, and George.

Let us now consider what happens if we ask the function in Figure 5.14 to search the list Alice, Carol, Evelyn, Fred, and George for the entry David. This time the original copy of the function selects Evelyn as its test entry and concludes that the target value must reside in the preceding portion of the list. It therefore

Figure 5.15 Recursively Searching



```
def Search (List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

```
def Search (List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

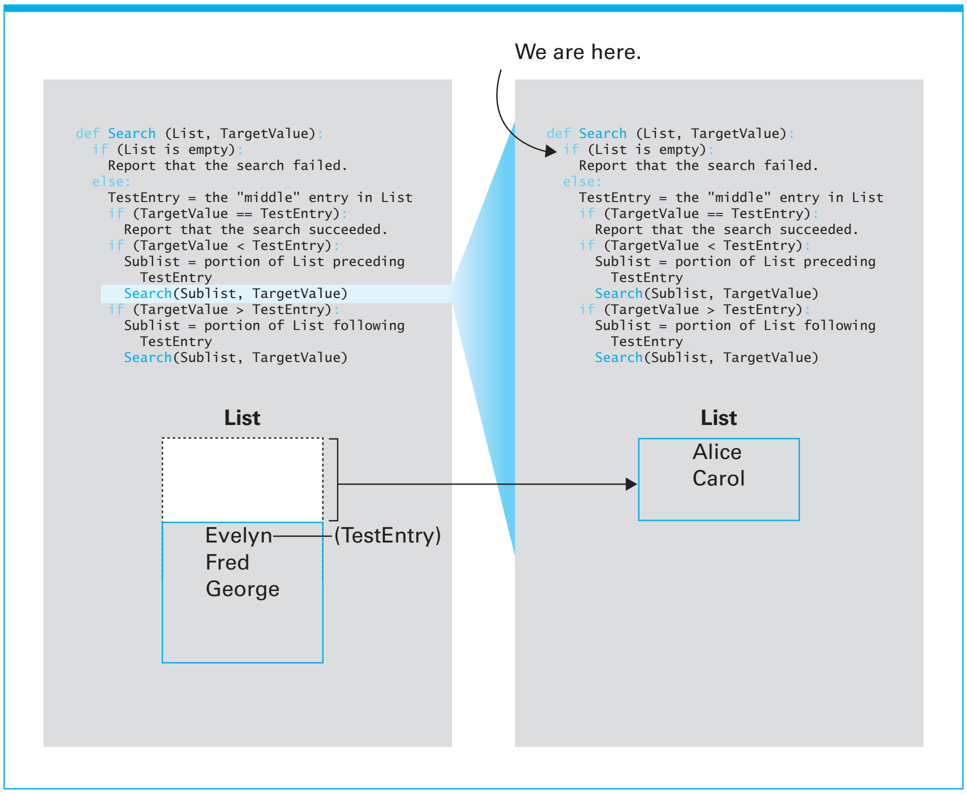
requests another copy of the function to search the list of entries appearing in front of Evelyn—that is, the two-entry list consisting of Alice and Carol. At this stage our situation is as represented in Figure 5.16.

The second copy of the function selects Carol as its current entry and concludes that the target value must lie in the latter portion of its list. It then requests a third copy of the function to search the list of names following Carol in the list Alice and Carol. This sublist is empty, so the third copy of the function has the task of searching the empty list for the target value David. Our situation at this point is represented by Figure 5.17. The original copy of the function is charged with the task of searching the list Alice, Carol, Evelyn, Fred, and George, with the test entry being Evelyn; the second copy is charged with searching the list Alice and Carol, with its test entry being Carol; and the third copy is about to begin searching the empty list.

Of course, the third copy of the function quickly declares its search to be a failure and terminates. The completion of the third copy's task allows the second copy to continue its task. It notes that the search it requested was unsuccessful, declares its own task to be a failure, and terminates. This report is what the original copy of the function has been waiting for, so it can now proceed. Since the search it requested failed, it declares its own search to have failed and terminates. Our routine has correctly concluded that David is not contained in the list Alice, Carol, Evelyn, Fred, and George.

In summary, if we were to look back at the previous examples, we could see that the process employed by the algorithm represented in Figure 5.14 is to

Figure 5.16 Second Recursive Search, First Snapshot



```
def Search (List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

```
def Search (List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

repeatedly divide the list in question into two smaller pieces in such a way that the remaining search can be restricted to only one of these pieces. This divide-by-two approach is the reason why the algorithm is known as the **binary search.**

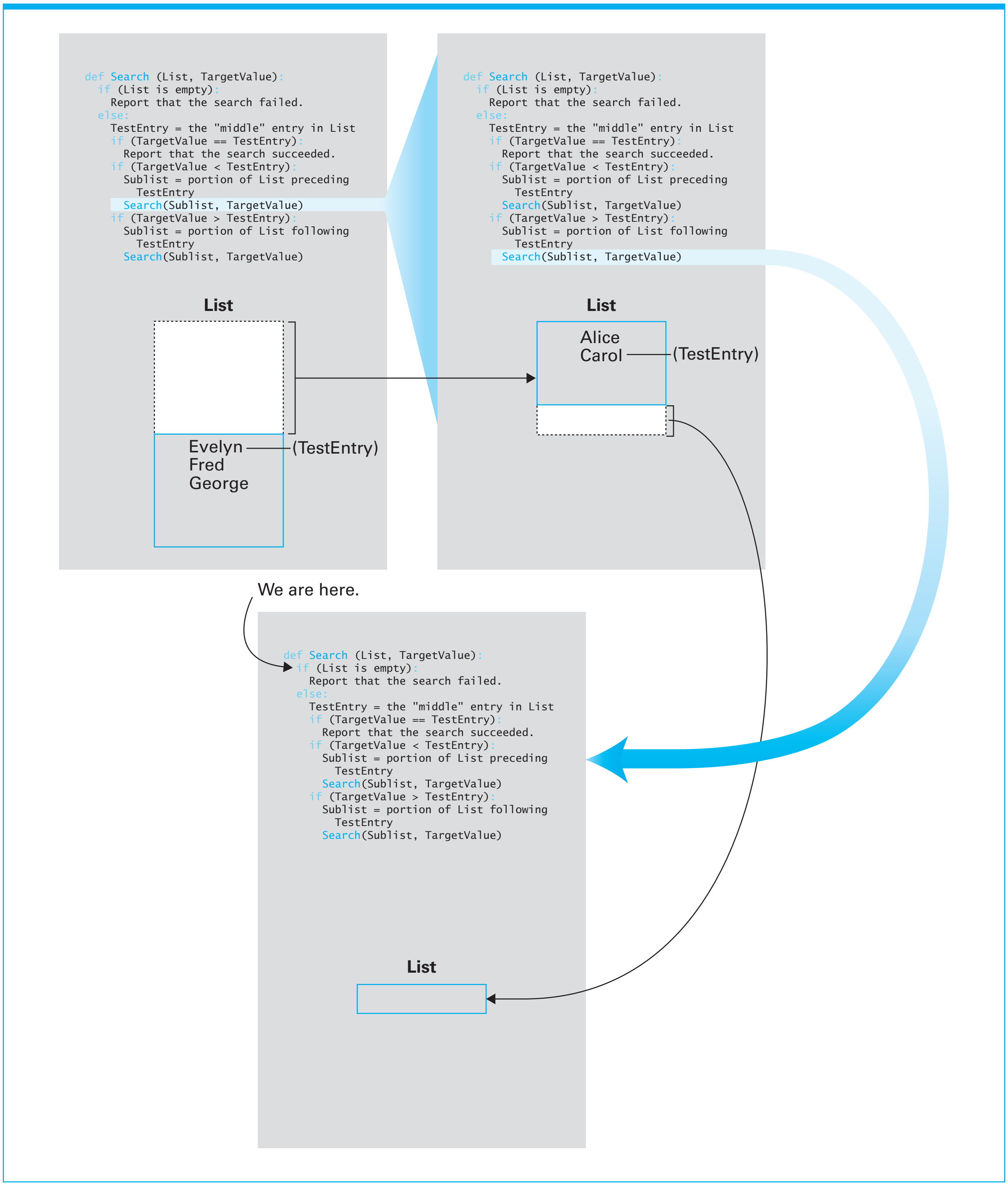
Recursive Control

The binary search algorithm is similar to the sequential search in that each algorithm requests the execution of a repetitive process. However, the implementation of this repetition is significantly different. Whereas the sequential search involves a circular form of repetition, the binary search executes each stage of the repetition as a subtask of the previous stage. This technique is known as **recursion.**

As we have seen, the illusion created by the execution of a recursive function is the existence of multiple copies of the function, each of which is called an activation of the function. These activations are created dynamically in a telescoping manner and ultimately disappear as the algorithm advances. Of those activations existing at any given time, only one is actively progressing. The others are effectively in limbo, each waiting for another activation to terminate before it can continue.

Being a repetitive process, recursive systems are just as dependent on proper control as are loop structures. Just as in loop control, recursive systems are dependent on testing for a termination condition and on a design that ensures

Figure 5.17 Second Recursive Search, Second Snapshot



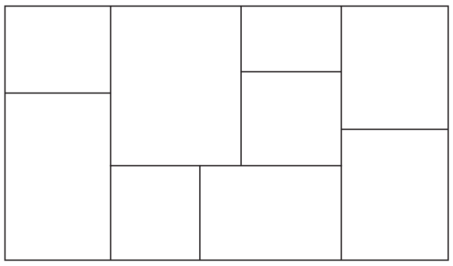
```
def Search (List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

```
def Search (List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

```
def Search (List, TargetValue):
  if (List is empty):
    Report that the search failed.
  else:
    TestEntry = the "middle" entry in List
    if (TargetValue == TestEntry):
      Report that the search succeeded.
    if (TargetValue < TestEntry):
      Sublist = portion of List preceding
        TestEntry
      Search(Sublist, TargetValue)
    if (TargetValue > TestEntry):
      Sublist = portion of List following
        TestEntry
      Search(Sublist, TargetValue)
```

Recursive Structures in Art

The following recursive function can be applied to a rectangular canvas to produce drawings of the style of the Dutch painter Piet Mondrian (1872–1944), who produced paintings in which the rectangular canvas was divided into successively smaller rectangles. Try following the function yourself to produce drawings similar to the one shown. Begin by applying the function to a rectangle representing the canvas on which you are working. (If you are wondering whether the algorithm represented by this function is an algorithm according to the definition in Section 5.1, your suspicions are well-founded. It is, in fact, an example of a nondeterministic algorithm since there are places at which the person or machine following the function is asked to make "creative" decisions. Perhaps this is why Mondrian's results are considered art while ours are not.)



```
def Mondrian (Rectangle):
  if (the size of Rectangle is too large for your artistic taste):
     divide Rectangle into two smaller rectangles.
     apply the function Mondrian to one of the smaller rectangles.
     apply the function Mondrian to the other smaller rectangle.
```

this condition will be reached. In fact, proper recursive control involves the same three ingredients—initialization, modification, and test for termination—that are required in loop control.

In general, a recursive function is designed to test for the termination condition (often called the **base case** or **degenerative case**) before requesting further activations. If the termination condition is not met, the routine creates another activation of the function and assigns it the task of solving a revised problem that is closer to the termination condition than that assigned to the current activation. However, if the termination condition is met, a path is taken that causes the current activation to terminate without creating additional activations.

Let us see how the initialization and modification phases of repetitive control are implemented in our binary search function of Figure 5.14. In this case, the creation of additional activations is terminated once the target value is found or the task is reduced to that of searching an empty list. The process is initialized implicitly by being given an initial list and a target value. From this initial configuration the function modifies its assigned task to that of searching a smaller list. Since the original list is of finite length and each modification step reduces the length of the list in question, we are assured that the target value ultimately is found or the task is reduced to that of searching the empty list. We can therefore conclude that the repetitive process is guaranteed to cease.

Finally, since both loop and recursive control structures are ways to cause the repetition of a set of instructions, we might ask whether they are equivalent in power. That is, if an algorithm were designed using a loop structure, could another algorithm using only recursive techniques be designed that would solve the same problem and vice versa? Such questions are important in computer science because their answers ultimately tell us what features should be provided in a programming language in order to obtain the most powerful programming system possible. We will return to these ideas in Chapter 12 where we consider some of the more theoretical aspects of computer science and its mathematical foundations. With this background, we will then be able to prove the equivalence of iterative and recursive structures in Appendix E.

Questions & Exercises

1. What names are interrogated by the binary search (Figure 5.14) when searching for the name Joe in the list Alice, Brenda, Carol, Duane, Evelyn, Fred, George, Henry, Irene, Joe, Karl, Larry, Mary, Nancy, and Oliver?
2. What is the maximum number of entries that must be interrogated when applying the binary search to a list of 200 entries? What about a list of 100,000 entries?
3. What sequence of numbers would be printed by the following recursive function if we started it with N assigned the value 1?

```
def Exercise (N):
  print(N)
  if (N < 3):
    Exercise(N + 1)
  print(N)
```

4. What is the termination condition in the recursive function of question 3?

5.6 Efficiency and Correctness

In this section we introduce two topics that constitute important research areas within computer science. The first of these is algorithm efficiency, and the second is algorithm correctness.

Algorithm Efficiency

Even though today's machines are capable of executing millions or billions of instructions each second, efficiency remains a major concern in algorithm design. Often the choice between efficient and inefficient algorithms can make the difference between a practical solution to a problem and an impractical one.

Let us consider the problem of a university registrar faced with the task of retrieving and updating student records. Although the university has an actual enrollment of approximately 10,000 students during any one semester, its "current student" file contains the records of more than 30,000 students who are considered current in the sense that they have registered for at least one course in the past few years but have not completed a degree. For now, let us assume that these records are stored in the registrar's computer as a list ordered by student identification numbers. To find any student record, the registrar would therefore search this list for a particular identification number.

We have presented two algorithms for searching such a list: the sequential search and the binary search. Our question now is whether the choice between these two algorithms makes any difference in the case of the registrar. We consider the sequential search first.

Given a student identification number, the sequential search algorithm starts at the beginning of the list and compares the entries found to the identification number desired. Not knowing anything about the source of the target value, we cannot conclude how far into the list this search must go. We can say, though, that after many searches we expect the average depth of the searches to be halfway through the list; some will be shorter, but others will be longer. Thus, we estimate that over a period of time, the sequential search will investigate roughly 15,000 records per search. If retrieving and checking each record for its identification number requires 10 milliseconds (10 one-thousandths of a second), such a search would require an average of 150 seconds or 2.5 minutes—an unbearably long time for the registrar to wait for a student's record to appear on a computer screen. Even if the time required to retrieve and check each record were reduced to only 1 millisecond, the search would still require an average of 15 seconds, which is still a long time to wait.

In contrast, the binary search proceeds by comparing the target value to the middle entry in the list. If this is not the desired entry, then at least the remaining search is restricted to only half of the original list. Thus, after interrogating the middle entry in the list of 30,000 student records, the binary search has at most 15,000 records still to consider. After the second inquiry, at most 7,500 remain, and after the third retrieval, the list in question has dropped to no more than 3,750 entries. Continuing in this fashion, we see that the target record will be found after retrieving at most 15 entries from the list of 30,000 records. Thus, if each of these retrievals can be performed in 10 milliseconds, the process of searching for a particular record requires only 0.15 of a second—meaning that access to any particular student record will appear to be instantaneous from the registrar's point of view. We conclude that the choice between the sequential search algorithm and the binary search algorithm would have a significant impact in this application.

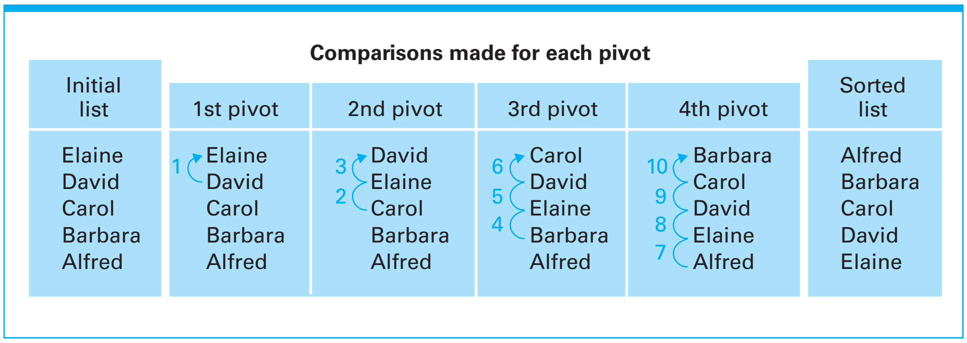
This example indicates the importance of the area of computer science known as algorithm analysis that encompasses the study of the resources, such as time or storage space, that algorithms require. A major application of such studies is the evaluation of the relative merits of alternative algorithms. Algorithm analysis often involves best-case, worst-case, and average-case scenarios. In our example, we performed an average-case analysis of the sequential search algorithm and a worst-case analysis of the binary search algorithm in order to estimate the time required to search through a list of 30,000 entries. In general such analysis is performed in a more generic context. That is, when considering algorithms for searching lists, we do not focus on a list of a particular length, but instead try to identify a formula that would indicate the algorithm's performance for lists of arbitrary lengths. It is not difficult to generalize our previous reasoning to lists of arbitrary lengths. In particular, when applied to a list with n entries, the sequential search algorithm will interrogate an average of $n/2$ entries, whereas the binary search algorithm will interrogate at most $\log_2 n$ entries in its worst-case scenario. ($\log_2 n$ represents the base two logarithm of n. Unless otherwise stated, computer scientists usually mean base two when talking about logorithms.)

Let us analyze the insertion sort algorithm (summarized in Figure 5.11) in a similar manner. Recall that this algorithm involves selecting a list entry, called the pivot entry, comparing this entry to those preceding it until the proper place for the pivot is found, and then inserting the pivot entry in this place. Since the activity of comparing two entries dominates the algorithm, our approach will be to count the number of such comparisons that are performed when sorting a list whose length is n.

The algorithm begins by selecting the second list entry to be the pivot. It then progresses by picking successive entries as the pivot until it has reached the end of the list. In the best possible case, each pivot is already in its proper place, and thus it needs to be compared to only a single entry before this is discovered. Thus, in the best case, applying the insertion sort to a list with n entries requires n - 1 comparisons. (The second entry is compared to one entry, the third entry to one entry, and so on.)

In contrast, the worst-case scenario is that each pivot must be compared to all the preceding entries before its proper location can be found. This occurs if the original list is in reverse order. In this case the first pivot (the second list entry) is compared to one entry, the second pivot (the third list entry) is compared to two entries, and so on (Figure 5.18). Thus the total number of comparisons when sorting a list of n entries is $1 + 2 + 3 + \ldots + (n - 1)$, which is equivalent to $(1/2)(n^2 - n)$. In particular, if the list contained 10 entries, the worst-case scenario of the insertion sort algorithm would require 45 comparisons.

Figure 5.18 Applying the insertion sort in a worst-case situation.



In the average case of the insertion sort, we would expect each pivot to be compared to half of the entries preceding it. This results in half as many comparisons as were performed in the worst case, or a total of $(^1/_4)(n^2 - n)$ comparisons to sort a list of n entries. If, for example, we use the insertion sort to sort a variety of lists of length 10, we expect the average number of comparisons per sort to be 22.5.

The significance of these results is that the number of comparisons made during the execution of the insertion sort algorithm gives an approximation of the amount of time required to execute the algorithm. Using this approximation, Figure 5.19 shows a graph indicating how the time required to execute the insertion sort algorithm increases as the length of the list increases. This graph is based on our worst-case analysis of the algorithm, where we concluded that sorting a list of length n would require at most $(^1/_2)(n^2 - n)$ comparisons between list entries. On the graph, we have marked several list lengths and indicated the time required in each case. Notice that as the list lengths increase by uniform increments, the time required to sort the list increases by increasingly greater amounts. Thus the algorithm becomes less efficient as the size of the list increases.

Let us apply a similar analysis to the binary search algorithm. Recall that we concluded that searching a list with n entries using this algorithm would require interrogating at most $log_2\ n$ entries, which again gives an approximation to the amount of time required to execute the algorithm for various list sizes. Figure 5.20 shows a graph based on this analysis on which we have again marked several list lengths of uniformly increasing size and identified the time required by the algorithm in each case. Note that the time required by the algorithm increases by decreasing increments. That is, the binary search algorithm becomes more efficient as the size of the list increases.

Figure 5.19 Graph of the worst-case analysis of the insertion sort algorithm

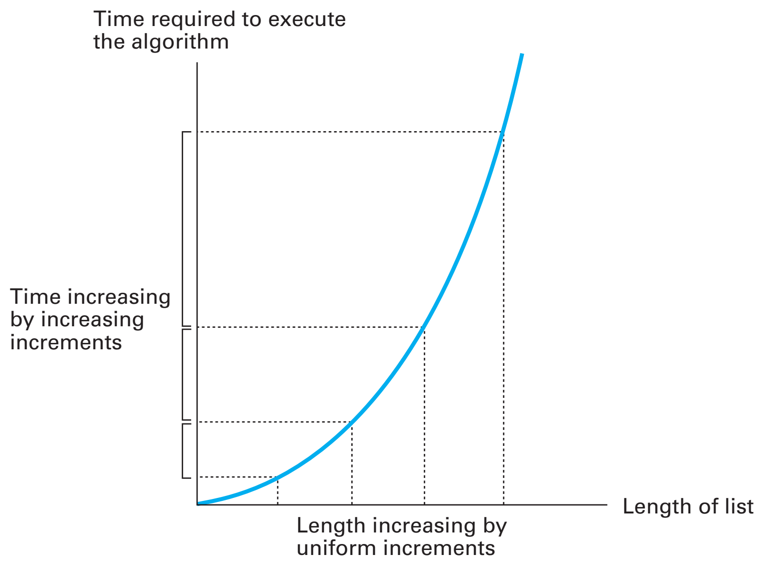
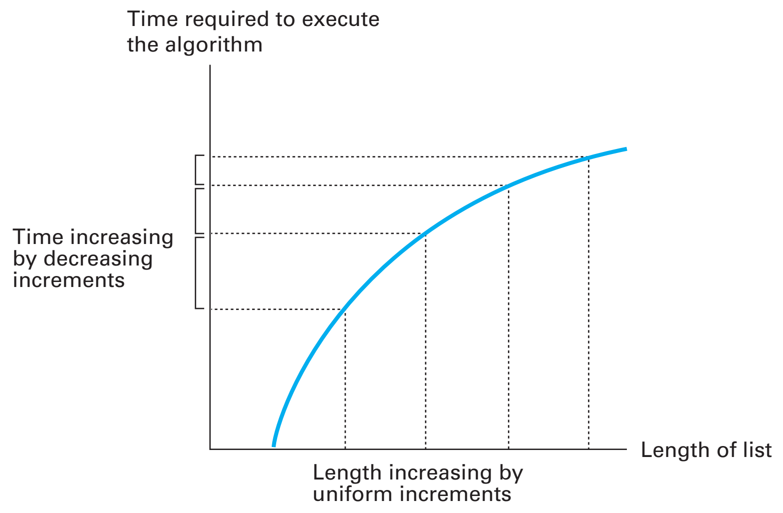


Figure 5.20 Graph of the worst-case analysis of the binary search algorithm



The distinguishing factor between Figures 5.19 and 5.20 is the general shape of the graphs involved. This general shape reveals how well an algorithm should be expected to perform for larger and larger inputs. Moreover, the general shape of a graph is determined by the type of the expression being represented rather than the specifics of the expression—all linear expressions produce a straight line; all quadratic expressions produce a parabolic curve; all logarithmic expressions produce the logarithmic shape shown in Figure 5.20. It is customary to identify a shape with the simplest expression that produces that shape. In particular, we identify the parabolic shape with the expression n^2 and the logarithmic shape with the expression $log_2\ n$.

Since the shape of the graph obtained by comparing the time required for an algorithm to perform its task to the size of the input data reflects the efficiency characteristics of the algorithm, it is common to classify algorithms according to the shapes of these graphs—normally based on the algorithm's worst-case analysis. The notation used to identify these classes is sometimes called **big-theta notation.** All algorithms whose graphs have the shape of a parabola, such as the insertion sort, are put in the class represented by $\Theta(n^2)$ (read "big theta of *n* squared"); all algorithms whose graphs have the shape of a logarithmic expression, such as the binary search, fall in the class represented by $\Theta(log_2\ n)$ (read "big theta of *log n*"). Knowing the class in which a particular algorithm falls allows us to predict its performance and to compare it against other algorithms that solve the same problem. Two algorithms in $\Theta(n^2)$ will exhibit similar changes in time requirements as the size of the inputs increases. Moreover, the time requirements of an algorithm in $\Theta(log_2\ n)$ will not expand as rapidly as that of an algorithm in $\Theta(n^2)$.

Software Verification

Recall that the fourth phase in Polya's analysis of problem solving (Section 5.3) is to evaluate the solution for accuracy and for its potential as a tool for solving other problems. The significance of the first part of this phase is exemplified by the following example:

> A traveler with a gold chain of seven links must stay in an isolated hotel for seven nights. The rent each night consists of one link from the chain. What is the fewest number of links that must be cut so that the traveler can pay the hotel one link of the chain each morning without paying for lodging in advance?

To solve this problem we first realize that not every link in the chain must be cut. If we cut only the second link, we could free both the first and second links from the other five. Following this insight, we are led to the solution of cutting only the second, fourth, and sixth links in the chain, a process that releases each link while cutting only three (Figure 5.21). Furthermore, any fewer cuts leaves two links connected, so we might conclude that the correct answer to our problem is three.

Upon reconsidering the problem, however, we might make the observation that when only the third link in the chain is cut, we obtain three pieces of chain of lengths one, two, and four (Figure 5.22). With these pieces we can proceed as follows:

First morning: Give the hotel the single link.

Second morning: Retrieve the single link and give the hotel the two-link piece.

Third morning: Give the hotel the single link.

Fourth morning: Retrieve the three links held by the hotel and give the hotel the four-link piece.

Fifth morning: Give the hotel the single link.

Sixth morning: Retrieve the single link and give the hotel the double-link piece.

Seventh morning: Give the hotel the single link.

Consequently, our first answer, which we thought was correct, is incorrect. How, then, can we be sure that our new solution is correct? We might argue as follows: Since a single link must be given to the hotel on the first morning, at least one link of the chain must be cut, and since our new solution requires only one cut, it must be optimal.

Translated into the programming environment, this example emphasizes the distinction between a program that is believed to be correct and a program that is

Figure 5.21 Separating the chain using only three cuts

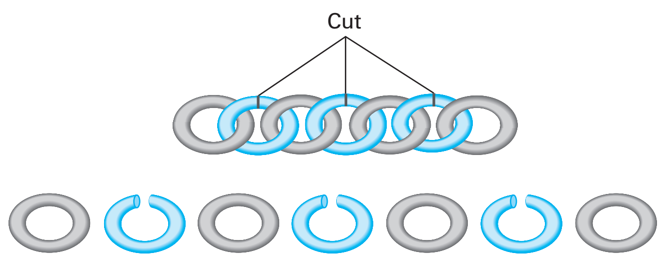
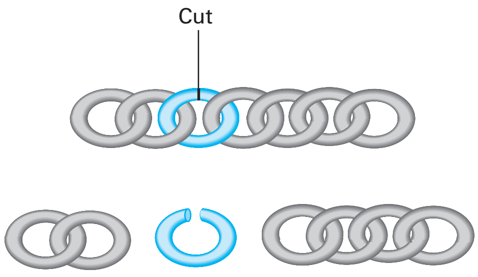


Figure 5.22 Solving the problem with only one cut



correct. The two are not necessarily the same. The data processing community is rich in horror stories involving software that although "known" to be correct, still failed at a critical moment because of some unforeseen situation. Verification of software is therefore an important undertaking, and the search for efficient verification techniques constitutes an active field of research in computer science.

A major line of research in this area attempts to apply the techniques of formal logic to prove the correctness of a program. That is, the goal is to apply formal logic to prove that the algorithm represented by a program does what it is intended to do. The underlying thesis is that by reducing the verification process to a formal procedure, one is protected from the inaccurate conclusions that might be associated with intuitive arguments, as was the case in the gold chain problem. Let us consider this approach to program verification in more detail.

Just as a formal mathematical proof is based on axioms (geometric proofs are often founded on the axioms of Euclidean geometry, whereas other proofs are based on the axioms of set theory), a formal proof of a program's correctness is based on the specifications under which the program was designed. To prove that a program correctly sorts lists of names, we are allowed to begin with the assumption that the program's input is a list of names, or if the program is designed to compute the average of one or more positive numbers, we assume that the input does, in fact, consist of one or more positive numbers. In short, a

Beyond Verification of Software

Verification problems, as discussed in the text, are not unique to software. Equally important is the problem of confirming that the hardware that executes a program is free of flaws. This involves the verification of circuit designs as well as machine construction. Again, the state of the art relies heavily on testing, which, as in the case of software, means that subtle errors can find their way into finished products. Records indicate that the Mark I, constructed at Harvard University in the 1940s, contained wiring errors that were not detected for many years. In the 1990s, a flaw was discovered in the floating- point portion of the early Pentium microprocessors that caused it to calculate the wrong answer when certain numbers were divided. In both of these cases, the error was detected before serious consequences developed.

proof of correctness begins with the assumption that certain conditions, called **preconditions,** are satisfied at the beginning of the program's execution.

The next step in a proof of correctness is to consider how the consequences of these preconditions propagate through the program. For this purpose, researchers have analyzed various program structures to determine how a statement, known to be true before the structure is executed, is affected by executing the structure. As a simple example, if a certain statement about the value of Y is known to hold prior to executing the instruction

```
X = Y
```

then that same statement can be made about X after the instruction has been executed. More precisely, if the value of Y is not 0 before the instruction is executed, then we can conclude that the value of X will not be 0 after the instruction is executed.

A slightly more involved example occurs in the case of an if-else structure such as

```
if (condition):
  instruction A
else:
  instruction B
```

Here, if some statement is known to hold before execution of the structure, then immediately before executing `instruction A`, we know that both that statement and the condition tested are true, whereas if `instruction B` is to be executed, we know the statement and the negation of the condition tested must hold.

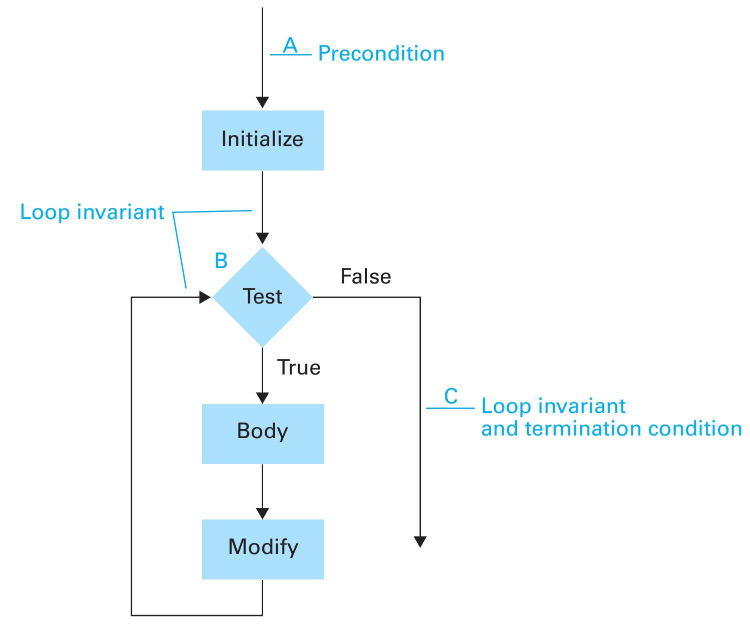
Following rules such as these, a proof of correctness proceeds by identifying statements, called **assertions,** that can be established at various points in the program. The result is a collection of assertions, each being a consequence of the program's preconditions and the sequence of instructions that lead to the point in the program at which the assertion is established. If the assertion so established at the end of the program corresponds to the desired output specifications (which are called **postconditions**), we conclude that the program is correct.

As an example, consider the typical `while` loop structure represented in Figure 5.23. Suppose, as a consequence of the preconditions given at point A, we can establish that a particular assertion is true each time the test for termination is performed (point B) during the repetitive process. (An assertion at a point in a loop that is true every time that point in the loop is reached is known as a **loop invariant.**) Then, if the repetition ever terminates, execution moves to point C, where we can conclude that both the loop invariant and the termination condition hold. (The loop invariant still holds because the test for termination does not alter any values in the program, and the termination condition holds because otherwise the loop does not terminate.) If these combined statements imply the desired postconditions, our proof of correctness can be completed merely by showing that the initialization and modification components of the loop ultimately lead to the termination condition.

You should compare this analysis to our example of the insertion sort shown in Figure 5.11. The outer loop in that program is based on the loop invariant

> Each time the test for termination is performed, the entries in the list from position 1 through position $N-1$ are sorted

Figure 5.23 The assertions associated with a typical while structure



and the termination condition is

The value of *N* is greater than the length of the list.

Thus, if the loop ever terminates, we know that both conditions must be satisfied, which implies that the entire list would be sorted.

Progress in the development of program verification techniques continues to be challenging. However, advancements are being made. One of the more significant is found in the programming language SPARK, which is closely related to the more popular language Ada. (Ada is one of the languages from which we will draw examples in the next chapter.) In addition to allowing programs to be expressed in a high-level form such as our pseudocode, SPARK gives programmers a means of including assertions such as preconditions, postconditions, and loop invariants within the program. Thus, a program written in SPARK contains not only the algorithm to be applied but also the information required for the application of formal proof-of-correctness techniques. To date, SPARK has been used successfully in numerous software development projects involving critical software applications, including secure software for the U.S. National Security Agency, internal control software used in Lockheed Martin's C130J Hercules aircraft, and critical rail transportation control systems.

In spite of successes such as SPARK, formal program verification techniques have not yet found widespread usage, and thus most of today's software is "verified" by testing—a process that is shaky at best. After all, verification by testing proves nothing more than that the program performs correctly for the cases under which it was tested. Any additional conclusions are merely projections. The errors contained in a program are often consequences of subtle oversights that are easily overlooked during testing as well as development. Consequently errors in a program, just as our error in the gold chain problem, can, and often do, go undetected, even though significant effort may be exerted to avoid it. A dramatic example occurred at AT&T: An error in the software controlling 114 switching stations went undetected from the software's installation in December 1989 until January 15, 1990, at which time a unique set of circumstances caused approximately five million calls to be unnecessarily blocked over a nine-hour period.

Questions & Exercises

1. Suppose we find that a machine programmed with our insertion sort algorithm requires an average of one second to sort a list of 100 names. How long do you estimate it takes to sort a list of 1,000 names? How about 10,000 names?
2. Give an example of an algorithm in each of the following classes: $\Theta(log_2\ n)$, $\Theta(n)$, and $\Theta(n^2)$.
3. List the classes $\Theta(n^2)$, $\Theta(log_2\ n)$, $\Theta(n)$, and $\Theta(n^3)$ in decreasing order of efficiency.
4. Consider the following problem and a proposed answer. Is the proposed answer correct? Why or why not?

   Problem:—Suppose a box contains three cards. One of three cards is painted black on both sides, one is painted red on both sides, and the third is painted red on one side and black on the other. One of the cards is drawn from the box, and you are allowed to see one side of it. What is the probability that the other side of the card is the same color as the side you see?

   Proposed answer:—One-half. Suppose the side of the card you can see is red. (The argument would be symmetric with this one if the side were black.) Only two cards among the three have a red side. Thus the card you see must be one of these two. One of these two cards is red on the other side, while the other is black. Thus the card you can see is just as likely to be red on the other side as it is to be black.
5. The following program segment is an attempt to compute the quotient (forgetting any remainder) of two positive integers (a dividend and a divisor) by counting the number of times the divisor can be subtracted from the dividend before what is left becomes less than the divisor. For instance, 7/3 should produce 2 because 3 can be subtracted from 7 twice. Is the program correct? Justify your answer.

```
Count = 0
Remainder = Dividend
repeat:
  Remainder = Remainder – Divisor
  Count = Count + 1
  until (Remainder < Divisor)
Quotient = Count.
```

6. The following program segment is designed to compute the product of two nonnegative integers X and Y by accumulating the sum of X copies of Y— that is, 3 times 4 is computed by accumulating the sum of three 4s. Is the program correct? Justify your answer.

```
Product = Y
Count = 1
while (Count < X):
  Product = Product + Y
  Count = Count + 1
```

7. Assuming the precondition that the value associated with N is a positive integer, establish a loop invariant that leads to the conclusion that if the following routine terminates, then Sum is assigned the value 0 + 1 + . . . + N.

```
Sum = 0
K = 0
while (K < N):
  K = K + 1
  Sum = Sum + K
```

Provide an argument to the effect that the routine does in fact terminate.

8. Suppose that both a program and the hardware that executes it have been formally verified to be accurate. Does this ensure accuracy?

Chapter Review Problems

(Asterisked problems are associated with optional sections.)

1. Give an example of a set of steps that conforms to the informal definition of an algorithm given in the opening paragraph of Section 5.1 but does not conform to the formal definition given in Figure 5.1.
2. Explain the distinction between an ambiguity in a proposed algorithm and an ambiguity in the representation of an algorithm.
3. Describe how the use of primitives helps remove ambiguities in an algorithm's representation.
4. Select a subject with which you are familiar and design a pseudocode for giving directions in that subject. In particular, describe the primitives you would use and the syntax you would use to represent them. (If you are having trouble thinking of a subject, try sports, arts, or crafts.)
5. Does the following program represent an algorithm in the strict sense? Why or why not?

```
Count = 0
while (Count != 5):
  Count = Count + 2
```

6. In what sense do the following three steps not constitute an algorithm?

Step 1: Draw a circle with center coordinates (2, 5) and radius 3.

Step 2: Draw a circle with center coordinates (6, 5) and radius 5.

Step 3: Draw a line segment whose endpoints are at the intersections of the previous two circles.

7. Rewrite the following program segment using a `repeat` structure rather than a `while` structure. Be sure the new version prints the same values as the original. Initialization:

```
num = 0
while (num < 50):
  if (num is Odd)
    print(num is Odd)
  num = num + 1
```

8. Rewrite the following program segment using a `while` structure rather than a `repeat` structure. Be sure the new version prints the same values as the original.

```
num = 100
repeat:
  print(num)
  num = num - 1
  until (num > 0)
```

9. What must be done to translate a posttest loop expressed in the form

```
repeat:
  (. . . )
  until (. . . )
```

into an equivalent posttest loop expressed in the form

```
do:
  (. . . )
  while (. . . )
```

10. Design an algorithm that, when given an arrangement of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, rearranges the digits so that the new arrangement represents the next larger value that can be represented by these digits (or reports that no such rearrangement exists if no rearrangement produces a larger value). Thus 5647382901 would produce 5647382910.

11. Design an algorithm for finding all the factors of a positive integer. For example, in the case of the integer 12, your algorithm should report the values 1, 2, 3, 4, 6, and 12.

12. Design an algorithm for determining whether a particular year is a leap year. For example, the year 2000 was a leap year.

13. What is the difference between a formal programming language and a pseudocode?

14. What is the difference between syntax and semantics?

15. The following is an addition problem in traditional base 10 notation. Each letter represents a different digit. What digit does each letter represent? How did you get your foot in the door?

```
   SEND
 + MORE
  MONEY
```

16. The following is a multiplication problem in traditional base 10 notation. Each letter represents a different digit. What digit does each letter represent? How did you get your foot in the door?

```
    XY
  × YX
    XY
   YZ
   WVY
```

17. The following is a multiplication problem in binary notation. Each letter represents a unique binary digit. Which letter represents 1 and which represents 0? Design an algorithm for solving problems like this.

```
   XYYX
 × X XY
  XYYXY
```

18. Four prospectors with only one lantern must walk through a mine shaft. At most, two prospectors can travel together and any prospector in the shaft must be with the lantern. The prospectors, named Andrews, Blake, Johnson, and Kelly, can walk through the shaft in one minute, two minutes, four minutes, and eight minutes, respectively. When two walk together they travel at the speed of the slower prospector. How can all four prospectors get through the mine shaft in only 15 minutes? After you have solved this problem, explain how you got your foot in the door.

19. Starting with a large wine glass and a small wine glass, fill the small glass with wine and then pour that wine into the large glass. Next, fill the small glass with water and pour

some of that water into the large glass. Mix the contents of the large glass, and then pour the mixture back into the small glass until the small glass is full. Will there be more water in the large glass than there is wine in the small glass? After you have solved this problem, explain how you got your foot in the door.

20. Two bees, named Romeo and Juliet, live in different hives but have met and fallen in love. On a windless spring morning, they simultaneously leave their respective hives to visit each other. Their routes meet at a point 50 meters from the closest hive, but they fail to see each other and continue on to their destinations. At their destinations, they spend the same amount of time to discover that the other is not home and begin their return trips. On their return trips, they meet at a point that is 20 meters from the closest hive. This time they see each other and have a picnic lunch before returning home. How far apart are the two hives? After you have solved this problem, explain how you got your foot in the door.

21. Design an algorithm that, given two strings of characters, tests whether the second string is same as the first string or not.

22. The following algorithm is designed to print the beginning of what is known as the Fibonacci sequence. Identify the body of the loop. Where is the initialization step for the loop control? The modification step? The test step? What list of numbers is produced?

```
Last = 0
Current = 1
while (Current < 100):
  print(Current)
  Temp = Last
  Last = Current
  Current = Last + Temp
```

23. What sequence of numbers is printed by the following algorithm if the input value is 1?

```
def CodeWrite (num):
  While (num < 100):
    for (i = 2; i <= num/2; ++i)
    if (num % i == 0) flag = 1;
    if (flag == 0) print (num);
    num ++;
```

24. Modify the function CodeWrite in the preceding problem so that perfect square values are printed.

25. What letters are interrogated by the binary search (Figure 5.14) if it is applied to the list A, B, C, D, E, F, G, H, I, J, K, L, M, N, O when searching for the value J? What about searching for the value Z?

26. After performing many sequential searches on a list of 6,000 entries, what would you expect to be the average number of times that the target value would have been compared to a list entry? What if the search algorithm was the binary search?

27. Identify the termination condition in each of the following iterative statements.

a.
```
while (Count < 5):
  . . .
```
b.
```
repeat:
  . . .
  until (Count == 1)
```
c.
```
while ((Count < 5) and (Total < 56)):
  . . .
```

28. Identify the body of the following loop structure and count the number of times it will be executed. What happens if the test is changed to read "(Count != 6)"?

```
Count = 1
while (Count != 7):
  print(Count)
  Count = Count + 3
```

29. What problems do you expect to arise if the following program is implemented on a computer? (Hint: Remember the problem of round-off errors associated with floating-point arithmetic.)

```
Count = one_tenth
repeat:
  print(Count)
  Count = Count + one_tenth
  until (Count == 1)
```

30. Design a recursive version of the Euclidean algorithm (question 3 of Section 5.2).

31. Suppose we apply both Check1 and Check2 (defined next) to the input value 1. What is the difference in the printed output of the two routines?

```
def Check1 (num):
  if (num % 2 == 0):
    print(num)
    Check1(num + 1)
def Check2(num):
  if (num % 2 == 1):
    print(num)
    Check2(num + 1)
```

32. Identify the important constituents of the control mechanism in the routines of the previous problem. In particular, what condition causes the process to terminate? Where is the state of the process modified toward this termination condition? Where is the state of the control process initialized?

33. Identify the termination condition in the following recursive function.

```
def XXX (N):
  if (N == 5):
    XXX(N + 1)
```

34. Call the function CodeWrite (defined below) with the value 100 and record the values that are printed.

```
def CodeWrite (N):
  if (N > 0):
    print(N)
    CodeWrite(N / 2)
  print(N + 1)
```

35. Call the function MysteryPrint (defined below) with the value 2 and record the values that are printed.

```
def MysteryPrint (N):
  if (N > 0):
    print(N)
    MysteryPrint(N - 2)
  else:
    print(N)
    if (N > -1):
      MysteryPrint(N + 1)
```

36. Design an algorithm to generate the sequence of positive integers (in increasing order) whose only prime divisors are 2 and 3; that is, your program should produce the sequence 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, ... Does your program represent an algorithm in the strict sense?

37. Answer the following questions in terms of the list: 19, 37, 53, 71, 96, 137, 289, 374, 559, 797, 979.

a. Which search algorithm (sequential or binary) will find the number 137 more quickly?

b. Which search algorithm (sequential or binary) will find the number 19 more quickly?

c. Which search algorithm (sequential or binary) will detect the absence of the number 99 more quickly?

d. Which search algorithm (sequential or binary) will detect the absence of the number 111 more quickly?

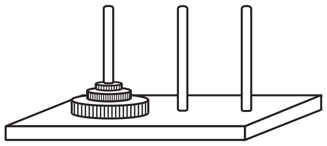
e. How many numbers will be interrogated when searching for the number 96 when using the sequential search? How many will be interrogated when using the binary search?

38. A positive integer is called an Armstrong number if the sum of the cubes of individual digits of the number is equal to that number itself. For example, the sum of cubes of individual digits of the number 153 is 1* 1 * 1 + 5 * 5 * 5 + 3 * 3 * 3 = 153. Hence, the number 153 is called an Armstrong number. Design an algorithm that checks whether a given number is an Armstrong number or not.

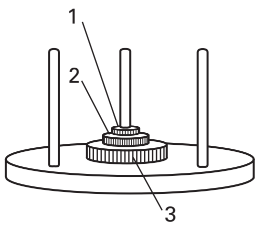
39. a. Suppose you must sort a list of ten integers in descending order, and you have already designed an algorithm that does this for six integers. Design an algorithm to sort ten integers by taking advantage of the previously designed algorithm.

b. Design a recursive algorithm to sort arbitrary lists of integers in descending order based on the technique used in (a).

40. The puzzle called the Towers of Hanoi consists of three pegs, one of which contains several rings stacked in order of descending diameter from bottom to top. The problem is to move the stack of rings to another peg. You are allowed to move only one ring at a time, and at no time is a ring to be placed on top of a smaller one. Observe that if the puzzle involved only one ring, it would be extremely easy. Moreover, when faced with the problem of moving several rings, if you could move all but the largest ring to another peg, the largest ring could then be placed on the third peg, and then the problem would be to move the remaining rings on top of it. Using this observation, develop a recursive algorithm for solving the Towers of Hanoi puzzle for an arbitrary number of rings.



41. Another approach to solving the Towers of Hanoi puzzle (question 40) is to imagine the pegs arranged on a circular stand with a peg mounted at each of the positions of 4, 8, and 12 o'clock. The rings, which begin on one of the pegs, are numbered 1, 2, 3, and so on, starting with the smallest ring being 1. Odd-numbered rings, when on top of a stack, are allowed to move clockwise to the next peg; likewise, even-numbered rings are allowed to move counterclockwise (as long as that move does not place a ring on a smaller one). Under this restriction, always move the largest-numbered ring that can be moved. Based on this observation, develop a nonrecursive algorithm for solving the Towers of Hanoi puzzle.



42. Develop two algorithms, one based on a loop structure and the other on a recursive structure, to print the daily salary of a worker who each day is paid twice the previous day's salary (starting with one penny for the first day's work) for a 30-day period. What problems relating to number storage are you likely to encounter if you implement your solutions on an actual machine?

43. Design an algorithm to find the square root of a positive number by starting with the number itself as the first guess and repeatedly producing a new guess from the previous one by averaging the previous guess with the result of dividing the original number by the previous guess. Analyze the control of this repetitive process. In particular, what condition should terminate the repetition?

44. Design an algorithm that lists all possible 4-digit integers made from digits 1 to 5 and are less than 5000.

45. Design an algorithm that, given a list of names, finds the longest name in the list. Use the `for` loop structure. Determine what your solution does if there are several "longest" names in the list. In particular, what would your algorithm do if all the names had the same length?

46. Design an algorithm that, given a list of nine or more names, sorts the list in alphabetical order and finds the name with the minimum number of characters.

47. Arrange the names Brenda, Doris, Raymond, Steve, Timothy, and William in an order that requires the least number of comparisons when sorted by the insertion sort algorithm (Figure 5.11).

48. What is the largest number of entries that are interrogated if the binary search algorithm (Figure 5.14) is applied to a list of 4,000 names? How does this compare to the sequential search (Figure 5.6)?

49. Use big-theta notation to classify the traditional grade school algorithms for addition and multiplication. That is, if asked to add two numbers each having n digits, how many individual additions must be performed? If requested to multiply two n-digit numbers, how many individual multiplications are required?

50. Sometimes a slight change in a problem can significantly alter the form of its solution. For example, find a simple algorithm for solving the following problem and classify it using big-theta notation:

> Divide a group of people into two disjoint subgroups (of arbitrary size) such that the difference in the total ages of the members of the two subgroups is as large as possible.

Now change the problem so that the desired difference is as small as possible and classify your approach to the problem.

51. From a given a list of 1000 integers from 1 to 1000, extract pairs of numbers whose product is 2424. How efficient is your approach to the problem?

52. Explain what will be the values of a and b if we call the function `CodeWrite` (defined below) with inputs 78 and 89.

```
def CodeWrite (a, b):
if (a > 0 and b > 0):
  a = a + b;
  a = a + b;
  a = a + b;
  print(a and b)
```

53. The following program segment is designed to compute the product of two nonnegative integers X and Y by accumulating the sum of X copies of Y; that is, 3 times 4 is computed by accumulating the sum of three 4s. Is the program segment correct? Explain your answer.

```
Product = 0
Count = 0
repeat:
  Product = Product + Y
  Count = Count + 1
  until (Count == X)
```

54. The following program segment is designed to report which of the positive integers X and Y is a divisor of the other. Is the program segment correct? Explain your answer.

```
if (X < Y)
if (Y % X == 0):
  print('X is Divisor of Y')
else if (X % Y == 0):
  print('Y is Divisor of X')
```

55. The following program segment is designed to find the smallest entry in a nonempty list of integers. Is it correct? Explain your answer.

```
TestValue = first list entry
CurrentEntry = first list entry
while (CurrentEntry is not the
    last entry):
  if (CurrentEntry < TestValue):
    TestValue = CurrentEntry
  CurrentEntry = the next list entry
```

56. a. Identify the preconditions for the sequential search as represented in Figure 5.6. Establish a loop invariant for the `while` structure in that program that, when combined with the termination condition, implies that upon termination of the loop, the algorithm will report success or failure correctly.

b. Give an argument showing that the `while` loop in Figure 5.6 does in fact terminate.

57. The following program segment is designed to calculate the factorial of a nonnegative number N using recursion. Is the program segment correct? Explain your answer. Modify this program segment to calculate the factorial without using recursion. Be sure that the output values do not change.

```
int fact (int N) {
if (N == 0)
return 1;
return N * fact (N - 1);
}
```

Social Issues

The following questions are intended as a guide to the ethical/social/legal issues associated with the field of computing. The goal is not merely to answer these questions. You should also consider why you answered as you did and whether your justifications are consistent from one question to the next.

1. Because it is currently impossible to verify completely the accuracy of complex programs, under what circumstances, if any, should the creator of a program be liable for errors?
2. Suppose you have an idea and develop it into a product that many people can use. Moreover, it has required a year of work and an investment of $50,000 to develop your idea into a form that is useful to the general public. In its

final form, however, the product can be used by most people without buying anything from you. What right do you have for compensation? Is it ethical to pirate computer software? What about music and motion pictures?

3. Suppose a software package is so expensive that it is totally out of your price range. Is it ethical to copy it for your own use? (After all, you are not cheating the supplier out of a sale because you would not have bought the package anyway.)
4. Ownership of rivers, forests, oceans, and so on has long been an issue of debate. In what sense should someone or some institution be given ownership of an algorithm?
5. Some people feel that new algorithms are discovered, whereas others feel that new algorithms are created. To which philosophy do you subscribe? Would the different points of view lead to different conclusions regarding ownership of algorithms and ownership rights?
6. Is it ethical to design an algorithm for performing an illegal act? Does it matter whether the algorithm is ever executed? Should the person who creates such an algorithm have ownership rights to that algorithm? If so, what should those rights be? Should algorithm ownership rights be dependent on the purpose of the algorithm? Is it ethical to advertise and circulate techniques for breaking security? Does it matter what is being broken into?
7. An author is paid for the motion picture rights to a novel even though the story is often altered in the film version. How much of a story has to change before it becomes a different story? What alterations must be made to an algorithm for it to become a different algorithm?
8. Educational software is now being marketed for children in the 18 months or younger age group. Proponents argue that such software provides sights and sounds that would otherwise not be available to many children. Opponents argue that it is a poor substitute for personal parent/child interaction. What is your opinion? Should you take any action based on your opinion without knowing more about the software? If so, what action?

Additional Reading

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Boston, MA: Addison-Wesley, 1974.

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

Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security.* Boston, MA: Addison-Wesley, 2003.

Gries, D. *The Science of Programming.* New York: Springer-Verlag, 1998.

Harbin, R. *Origami—the Art of Paper Folding.* London: Hodder Paperbacks, 1973.

Johnsonbaugh, R., and M. Schaefer. *Algorithms.* Upper Saddle River, NJ: Prentice-Hall, 2004.

Kleinberg, *Algorithm Design*, 2nd ed. Boston, MA: Addison-Wesley, 2014.

Knuth, D. E. *The Art of Computer Programming, Vol. 3,* 2nd ed. Boston, MA: Addison-Wesley, 1998.

Levitin, A. V. *Introduction to the Design and Analysis of Algorithms,* 3rd ed. Boston, MA: Addison-Wesley, 2011.

Polya, G. *How to Solve It.* Princeton, NJ: Princeton University Press, 1973.

Roberts, E. S. *Thinking Recursively.* New York: Wiley, 1986.