CHAPTER 1

An Introduction to Computer Science

CHAPTER TOPICS

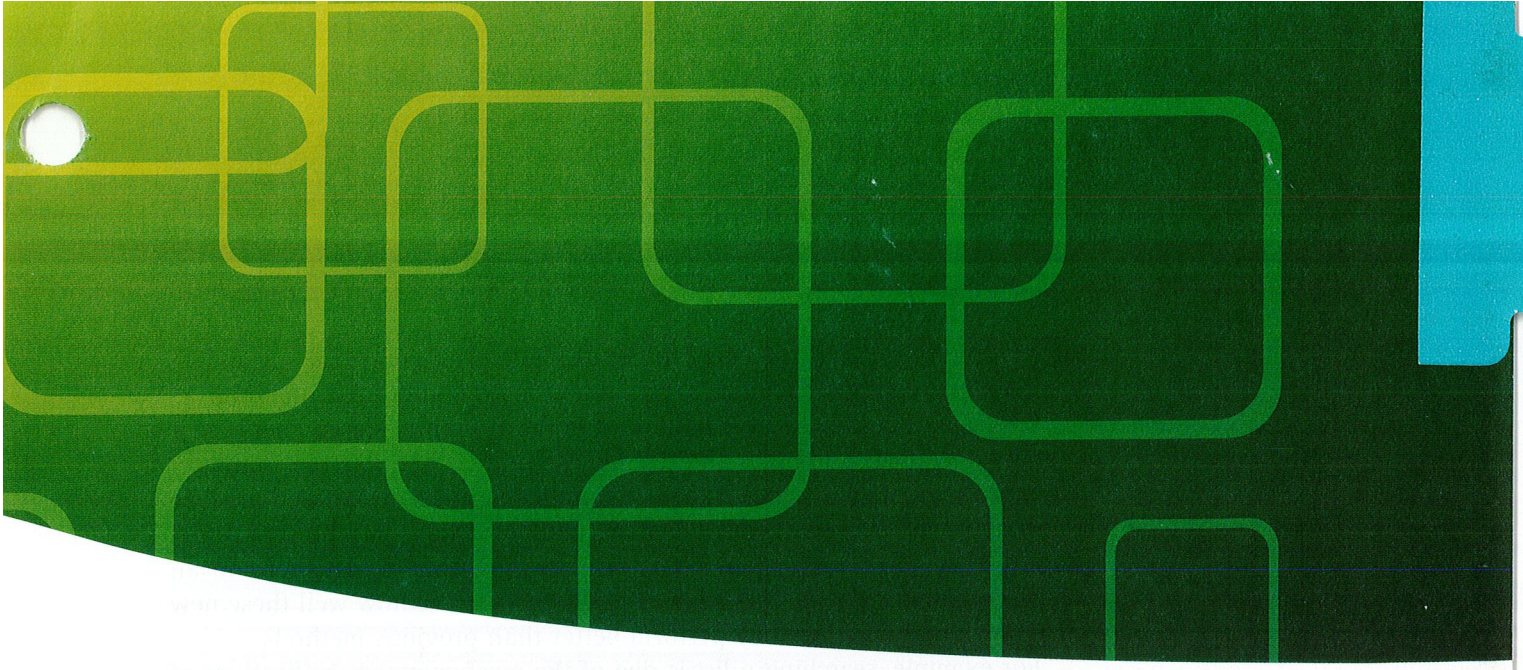
AFTER STUDYING THIS CHAPTER, YOU WILL BE ABLE TO:

- Understand the definition of the term algorithm
- Understand the formal definition of computer science
- Write down everyday algorithms
- Determine if an algorithm is ambiguous or not effectively computable
- Understand the roots of modern computer science in mathematics and mechanical machines
- Summarize the key points in the historical development of modern electronic computers

1.1 Introduction

This text is an invitation to learn about one of the youngest and most exciting scientific disciplines—*computer science*. Almost every day our newspapers, televisions, and electronic media carry reports of significant advances in computing, such as high-speed supercomputers that perform more than 90 quadrillion (10^{15}) mathematical operations per second; wireless networks that stream high-definition video and audio to the remotest corners of the globe in fractions of a second; minute computer chips that can be embedded into appliances, clothing, and even our bodies; and artificial intelligence systems that understand and respond to English language questions faster and more accurately than humans. The next few years will see technological breakthroughs that, until a few years ago, existed only in the minds of dreamers and science fiction writers. These are exciting times in computing, and our goal in this text is to provide you with an understanding of computer science and an appreciation for the diverse areas of research and study within this important field.

Although the average person can produce a reasonably accurate description of most scientific fields, even if he or she did not study the subject in school, many people do not have an intuitive understanding of the types of problems studied by computer science professionals. For example, you probably know that biology is the study of living organisms and that chemistry deals with the structure and composition of matter. However, you might not have the same fundamental understanding of the work that goes on in computer science. In fact, many people harbor one or more of the following common misconceptions about this field.



MISCONCEPTION 1: *Computer science is the study of computers.*

This apparently obvious definition is actually incorrect or, to put it more precisely, incomplete. For example, some of the earliest and most fundamental theoretical work in computer science took place from 1920 to 1940, many years before the development of the first computer system. (This pioneering work was initially considered a branch of logic and applied mathematics. Computer science did not come to be recognized as a separate and independent field of scientific study until the late 1950s and early 1960s.) Even today, there are branches of computer science quite distinct from the study of "real" machines. In *theoretical computer science*, for example, researchers study the logical and mathematical properties of problems and their solutions. Frequently, these researchers investigate problems not with actual computers but rather with *formal models* of computation, which are easier to study and analyze mathematically. Their work involves pencil and paper, not circuit boards and disks.

This distinction between computers and computer science was beautifully expressed by computer scientists Michael R. Fellows and Ian Parberry in an article in the journal *Computing Research News:*

> Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes, or chemistry is about beakers and test tubes. Science is not about tools. It is about how we use them and what we find out when we do.[1]

MISCONCEPTION 2: *Computer science is the study of how to write computer programs.*

Many people are introduced to computer science when learning to write programs in a language such as C++, Python, or Java. This almost universal

[1]Fellows, M. R., and Parberry, I. "Getting Children Excited About Computer Science," *Computing Research News,* vol. 5, no. 1 (January 1993).

use of programming as the entry to the discipline can create the misunderstanding that computer science is equivalent to computer programming.

Programming is extremely important to the discipline—researchers use it to study new ideas and build and test new solutions—but, like the computer itself, programming is a tool. When computer scientists design and analyze a new approach to solving a problem or create new ways to represent information, they often implement their ideas as programs to test them on an actual computer system. This enables researchers to see how well these new ideas work and whether they perform better than previous methods.

For example, searching a list is one of the most common applications of computers, and it is frequently applied to huge problems, such as finding one specific account among the approximately 63,000,000 active listings in the Social Security Administration database. A more efficient lookup method could significantly reduce the time that telephone-based customers must wait before receiving answers to questions regarding their accounts. Assume that we have designed what we believe to be a "new and improved" search technique. After analyzing it theoretically, we would study it empirically by writing a program to implement our new method, executing it on our computer, and measuring its performance. These tests would demonstrate under what conditions our new method is or is not faster than the search procedures currently in use.

In computer science, it is not simply the construction of a high-quality program that is important but also the methods it embodies, the services it provides, and the results it produces. It is possible to become so enmeshed in writing code and getting it to run that we forget that a program is only a means to an end, not an end in itself.

> MISCONCEPTION 3: *Computer science is the study of the uses and applications of computers and software.*

If one's introduction to computer science is not programming, then it might be a course on the application of computers and software. Such a course typically teaches the use of a number of popular packages, such as word processors, search engines, database systems, spreadsheets, presentation software, smartphone apps, and web browsers.

These packages are widely used by professionals in all fields. However, learning to use a software package is no more a part of computer science than driver's education is a branch of automotive engineering. A wide range of people *use* computer software, but it is the computer scientist who is responsible for *specifying*, *designing*, *building*, and *testing* these software packages as well as the computer systems on which they run.

These three misconceptions about computer science are not entirely wrong; they are just woefully incomplete. Computers, programming languages, software, and applications *are* part of the discipline of computer

science, but neither individually nor combined do they capture the richness and diversity of this field.

We have spent a good deal of time saying what computer science is *not*. What, then, is it? What are its basic concepts? What are the fundamental questions studied by professionals in this field? Is it possible to capture the breadth and scope of the discipline in a single definition? We answer these fundamental questions in the next section and, indeed, in the remainder of the text.

In the Beginning . . .

There is no single date that marks the beginning of computer science. Indeed, there are many "firsts" that could be used to mark this event. For example, some of the earliest theoretical work on the logical foundations of computer science occurred in the 1930s. The first general-purpose, electronic computers appeared during the period 1940–1946. (We will discuss the history of these early machines in Section 1.4.) These first computers were one-of-a-kind experimental systems that never moved outside the research laboratory. The first commercial machine, the UNIVAC I, did not make its appearance until March 1951, a date that marks the real beginning of the computer industry. The first high-level (i.e., based on natural language) programming language was FORTRAN. Some people mark its debut in 1957 as the beginning of the "software" industry. The appearance of these new machines and languages created new occupations, such as programmer, numerical analyst, and computer engineer. To address the intellectual needs of these workers, the first professional society for people in the field of computing, the Association for Computing Machinery (ACM), was established in 1947. (The ACM has more than 100,000 members and is the largest professional computer science society in the world. Its home page is *www.acm.org*.) To help meet the rapidly growing need for computer professionals, the first Department of Computer Science was established at Purdue University in October 1962. It awarded its first M.Sc. degree in 1964 and its first Ph.D. in computer science in 1966. An undergraduate program was established in 1967.

Thus, depending on what you consider the most important "first," the field of computer science is somewhere between 50 and 80 years old. Compared with such classic scientific disciplines as mathematics, physics, chemistry, and biology, computer science is the new kid on the block.

1.2 The Definition of Computer Science

There are many definitions of computer science, but the one that best captures the richness and breadth of ideas embodied in this branch of science was first proposed by professors Norman Gibbs and Allen Tucker.[2] According to their definition, the central concept in computer science is the algorithm. It is not possible to understand the field without a thorough understanding of this critically important idea.

DEFINITION

Computer science: the study of algorithms, including

1. Their formal and mathematical properties
2. Their hardware realizations
3. Their linguistic realizations
4. Their applications

The Gibbs and Tucker definition says that it is the task of the computer scientist to design and develop algorithms to solve a range of important problems. This design process includes the following operations:

- Studying the behavior of algorithms to determine if they are correct and efficient (their formal and mathematical properties)
- Designing and building computer systems that are able to execute algorithms (their hardware realizations)
- Designing programming languages and translating algorithms into these languages so that they can be executed by the hardware (their linguistic realizations)
- Identifying important problems and designing correct and efficient software packages to solve these problems (their applications)

Because it is impossible to appreciate this definition fully without knowing what an algorithm is, let's look more closely at this term. The Merriam-Webster dictionary (*www.merriam-webster.com/dictionary/*) defines the word *algorithm* as follows:

> *al • go • rithm n. A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step method for accomplishing some task.*

Informally, an algorithm is an ordered sequence of instructions that is guaranteed to solve a specific problem. It is a list that looks something like this:

Step 1: Do something

Step 2: Do something

Step 3: Do something

. .

. .

. .

Step N: Stop, you are finished

[2]Gibbs, N. E., and Tucker, A. B. "A Model Curriculum for a Liberal Arts Degree in Computer Science," *Comm. of the ACM*, vol. 29, no. 3 (March 1986).

If you are handed this list and carefully follow its instructions in the order specified, when you reach the end you will have solved the task at hand.

All the operations used to construct algorithms belong to one of only three categories:

1. Sequential operations. A sequential instruction carries out a single well-defined task. When that task is finished, the algorithm moves on to the next operation. Sequential operations are usually expressed as simple declarative sentences.
   - Add 1 cup of butter to the mixture in the bowl.
   - Subtract the amount of the check from the current account balance.
   - Set the value of x to 1.
2. Conditional operations. These are the "question-asking" instructions of an algorithm. They ask a question, and the next operation is then selected on the basis of the answer to that question.
   - If the mixture is too dry, then add one-half cup of water to the bowl.
   - If the amount of the check is less than or equal to the current account balance, then cash the check; otherwise, tell the person there are insufficient funds.
   - If x is not equal to 0, then set y equal to $1/x$; otherwise, print an error message that says you cannot perform division by 0.
3. Iterative operations. These are the "looping" instructions of an algorithm. They tell us not to go on to the next instruction but, instead, to go back and repeat the execution of a previous block of instructions.
   - Repeat the previous two operations until the mixture has thickened.
   - While there are still more checks to be processed, do the following five steps.
   - Repeat Steps 1, 2, and 3 until the value of y is equal to $+1$.

We use algorithms (although we don't call them that) all the time—whenever we follow a set of instructions to assemble a child's toy, bake a cake, balance a checkbook, or go through the college registration process. A good example of an algorithm used in everyday life is the set of instructions shown in Figure 1.1 for programming a DVR to record a collection of television shows. Note the three types of instructions in this algorithm: sequential (Steps 3, 4, 5, and 7), conditional (Steps 1 and 6), and iterative (Step 2).

Mathematicians use algorithms all the time, and much of the work done by early Greek, Roman, Persian, and Indian mathematicians involved the discovery of algorithms for important problems in geometry and arithmetic; an example is *Euclid's algorithm* for finding the greatest common divisor of two positive integers. (Exercise 10 at the end of the chapter presents this 2,300-year-old algorithm.) We also studied algorithms in elementary school, even if we didn't know it. For example, in the first grade we learned an algorithm for adding two numbers such as

$$\begin{array}{r} 47 \\ +25 \\ \hline 72 \end{array}$$

FIGURE 1.1

Step 1 If the clock and the calendar are not correctly set, then go to page 9 of the instruction manual and follow the instructions there before proceeding to Step 2

Step 2 Repeat Steps 3 through 6 for each program that you want to record

Step 3 Enter the channel number that you want to record and press the button labeled CHAN

Step 4 Enter the time that you want recording to start and press the button labeled TIME-START

Step 5 Enter the time that you want recording to stop and press the button labeled TIME-FINISH. This completes the programming of one show

Step 6 If you do not want to record anything else, press the button labeled END-PROG

Step 7 Turn off your DVR. Your DVR is now in TIMER mode, ready to record

Programming your DVR: An example of an algorithm

The instructions our teachers gave were as follows: First add the rightmost column of numbers (7 + 5), getting the value 12. Write down the 2 under the line and carry the 1 to the next column. Now move left to the next column, adding (4 + 2) and the previous carry value of 1 to get 7. Write this value under the line, producing the correct answer 72.

Although as children we learned this algorithm informally, it can, like the DVR instructions in Figure 1.1, be written formally as an explicit sequence of instructions. Figure 1.2 shows an algorithm for adding two positive m-digit numbers. It expresses formally the operations informally described previously. Again, note the three types of instructions used to construct the algorithm: sequential (Steps 1, 2, 4, 6, 7, 8, and 9), conditional (Step 5), and iterative (Step 3).

Even though it might not appear so, this is the same "decimal addition algorithm" that you learned in grade school; if you follow it rigorously, it is guaranteed to produce the correct result. Let's watch it work.

Add (47 + 25)
$m = 2$
$a_1 = 4$ $a_0 = 7$
$b_1 = 2$ $b_0 = 5$
} The input

STEP 1: *carry* = 0

STEP 2: $i = 0$

STEP 3: We now repeat Steps 4 through 6 while i is less than or equal to 1

First repetition of the loop (i has the value 0)

STEP 4: Add ($a_0 + b_0 + carry$), which is 7 + 5 + 0, so $c_0 = 12$

STEP 5: Because $c_0 \geq 10$, we reset **c_0 to 2** and reset *carry* to 1

FIGURE 1.2

Given: $m \geq 1$ and two positive numbers each containing m digits, $a_{m-1}\ a_{m-2} \ldots a_0$ and $b_{m-1}\ b_{m-2} \ldots b_0$

Wanted: $c_m\ c_{m-1}\ c_{m-2} \ldots c_0$, where $c_m\ c_{m-1}\ c_{m-2} \ldots c_0 = (a_{m-1}\ a_{m-2} \ldots a_0) + (b_{m-1}\ b_{m-2} \ldots b_0)$

Algorithm:

Step 1 Set the value of *carry* to 0

Step 2 Set the value of *i* to 0

Step 3 While the value of *i* is less than or equal to $m - 1$, repeat the instructions in Steps 4 through 6

Step 4 Add the two digits a_i and b_i to the current value of *carry* to get c_i

Step 5 If $c_i \geq 10$, then reset c_i to $(c_i - 10)$ and reset the value of *carry* to 1; otherwise, set the new value of *carry* to 0

Step 6 Add 1 to *i*, effectively moving one column to the left

Step 7 Set c_m to the value of *carry*

Step 8 Print out the final answer, $c_m\ c_{m-1}\ c_{m-2} \ldots c_0$

Step 9 Stop

Algorithm for adding two *m*-digit numbers

STEP 6: Reset *i* to (0 + 1) = 1. Because *i* is less than or equal to 1, go back to Step 4

Second repetition of the loop (*i* has the value 1)

STEP 4: Add (a_1 + b_1 + *carry*), which is 4 + 2 + 1, so $\mathbf{c_1 = 7}$

STEP 5: Because $c_1 < 10$, we reset *carry* to 0

STEP 6: Reset *i* to (1 + 1) = 2. Because *i* is greater than 1, we do not repeat the loop but instead go to Step 7

STEP 7: Set $\mathbf{c_2 = 0}$

STEP 8: Print out the answer $c_2\ c_1\ c_0$ = 072 (see the **boldface** values)

STEP 9: Stop

We have reached the end of the algorithm, and it has correctly produced the sum of the two numbers 47 and 25, the three-digit result 072. (A more clever algorithm would omit the unnecessary leading zero at the beginning of the number if the last carry value is a zero. That modification is an exercise—Exercise 6—at the end of the chapter.) Try working through the algorithm shown in Figure 1.2 with another pair of numbers to be sure that you understand exactly how it functions.

The addition algorithm shown in Figure 1.2 is a highly formalized representation of a technique that most people learned in the first or second grade and that virtually everyone knows how to do informally. Why would we take such a simple task as adding two numbers and express it in so

Abu Ja'far Muhammad ibn Musa Al-Khwarizmi (AD 780–850?)

The word *algorithm* is derived from the last name of Muhammad ibn Musa Al-Khwarizmi, a famous Persian mathematician and author from the eighth and ninth centuries. Al-Khwarizmi was a teacher at the House of Wisdom in Baghdad and the author of the book *Kitab al jabr w'al muqabala,* which in English means "The Concise Book of Calculation by Reduction." Written in AD 820, it is one of the earliest mathematical textbooks, and its title gives us the word *algebra* (the Arabic word *al jabr* means "reduction").

In AD 825, Al-Khwarizmi wrote another book about the base-10 positional numbering system that had recently been developed in India. In this book, he described formalized, step-by-step procedures for doing arithmetic operations, such as addition, subtraction, and multiplication, on numbers represented in this new decimal system, much like the addition algorithm diagrammed in Figure 1.2. In the twelfth century, this book was translated into Latin, introducing the base-10 Hindu–Arabic numbering system to Europe, and Al-Khwarizmi's name became closely associated with these formal numerical techniques. His last name was rendered as Algoritmi in Latin characters, and eventually the formalized procedures that he pioneered and developed became known as *algorithms* in his honor.

complicated a fashion? Why are formal algorithms so important in computer science? Because of the following fundamental idea:

If we can specify an algorithm to solve a problem, then we can automate its solution.

Once we have formally specified an algorithm, we can build a machine (or write a program or hire a person) to carry out the steps contained in the algorithm. The machine (or program or person) need not understand the concepts or ideas underlying the solution. It merely has to do Step 1, Step 2, Step 3, ... exactly as written. In computer science terminology, the machine, robot, person, or thing carrying out the steps of the algorithm is called a **computing agent**.

Thus, computer science can also be viewed as the *science of algorithmic problem solving.* Much of the research and development work in computer science involves discovering correct and efficient algorithms for a wide range of interesting problems, studying their properties, designing

programming languages into which those algorithms can be encoded, and designing and building computer systems that can automatically execute these algorithms in an efficient manner.

At first glance, it might seem that every problem can be solved algorithmically. However, you will learn in Chapter 12 the startling fact (first proved by the German logician Kurt Gödel in the early 1930s) that there are problems for which no generalized algorithmic solution can possibly exist. These problems are, in a sense, *unsolvable*. No matter how much time and effort is put into obtaining a solution, none will ever be found. Gödel's discovery, which staggered the mathematical world, effectively places a limit on the ultimate capabilities of computers and computer scientists.

There are also problems for which it is theoretically possible to specify an algorithm but a computing agent would take so long to execute it that the solution is essentially useless. For example, to get a computer to play winning chess, we could adopt a *brute force* approach. Given a board position as input, the computer would examine every legal move it could possibly make, then every legal response an opponent could make to each initial move, then every response it could select to that move, and so on. This analysis would continue until the game reached a win, lose, or draw position. With that information, the computer would be able to optimally choose its next move. If, for simplicity's sake, we assume that there are 40 legal moves from any given position on a chessboard, and it takes about 30 moves to reach a final conclusion, then the total number of board positions that our brute force program would need to evaluate in deciding its first move is

$$\underbrace{40 \times 40 \times 40 \times \ldots \times 40}_{30 \text{ times}} = 40^{30}, \text{ which is roughly } 10^{48}$$

If we use a supercomputer that evaluates 1 quadrillion (10^{15}) board positions per second, it would take about 30,000,000,000,000,000,000,000,000 years for the computer to make its first move! Obviously, a computer could not use a brute force technique to play a real chess game.

There also exist problems that we do not yet know *how* to solve algorithmically. Many of these involve tasks that require a degree of what we term "intelligence." For example, after only a few days a baby recognizes the face of his or her mother from among the many faces he or she sees. In a few months, the baby begins to develop coordinated sensory and motor control skills and can efficiently plan how to use them—how to get from the playpen to the toy on the floor without bumping into either the chair or the desk that is in the way. After a few years, the child begins to develop powerful language skills and abstract reasoning capabilities.

We take these abilities for granted, but, even though artificial intelligence research and implementation has made enormous strides in the past few years, with regard to the operations just mentioned—high-level problem solving, abstract reasoning, sophisticated natural-language understanding—the computer and software systems currently available in the marketplace have not yet achieved the intelligence level of an adult human being. The primary reason is that researchers do not yet know how to specify these operations

algorithmically. That is, they do not yet know how to specify a solution formally in a detailed step-by-step fashion. As humans, we are able to do them simply by using the "algorithms" in our heads. To appreciate this problem, imagine trying to describe algorithmically exactly what steps you follow when you are painting a picture, composing a love poem, or formulating a business plan.

Thus, algorithmic problem solving has many variations. Sometimes solutions do not exist; sometimes a solution is too inefficient to be of any use; sometimes a solution is not yet known. However, discovering an algorithmic solution has enormously important consequences. As we noted earlier, if we can create a correct and efficient algorithm to solve a problem, and if we encode it into a programming language, then we can take advantage of the speed and power of a computer system to automate the solution and produce the desired result. This is what computer science is all about.

1.3 Algorithms

DEFINITION

Algorithm: a well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time.

1.3.1 The Formal Definition of an Algorithm

The formal definition of an algorithm is rather imposing and contains a number of important ideas. Let's take it apart, piece by piece, and analyze each of its separate points.

...a well-ordered collection...

An algorithm is a collection of operations, and there must be a clear and unambiguous *ordering* to these operations. Ordering means that we know which operation to do first and precisely which operation to do next as each step is successfully completed. After all, we cannot expect a computing agent to carry out our instructions correctly if it is confused about which instruction it should be doing next.

Consider the following "algorithm" that was taken from the back of a shampoo bottle and is intended to be instructions on how to use the product.

Step 1: Wet hair

Step 2: Lather

Step 3: Rinse

Step 4: Repeat

At Step 4, what operations should be repeated? If we go back to Step 1, we will be unnecessarily wetting our hair. (It is presumably still wet from the previous operations.) If we go back to Step 3 instead, we will not be getting our hair any cleaner because we have not reused the shampoo. The Repeat instruction in Step 4 is ambiguous in that it does not clearly specify what to

do next. Therefore, it violates the well-ordered requirement of an algorithm. (It also has a second and even more serious problem—it never stops! We will have more to say about this second problem shortly.) Statements such as

- Go back and do it again. (Do *what* again?)
- Start over. (From *where*?)
- If you understand this material, you may skip ahead. (How *far*?)
- Do either Part 1 or Part 2. (How do I decide *which* one to do?)

are ambiguous and can leave us confused and unsure about what operation to do next. We must be extremely precise in specifying the order in which operations are to be carried out. One possible way is to number the steps of the algorithm and use these numbers to specify the proper order of execution. For example, the ambiguous operations just shown could be made more precise as follows:

- Go back to Step 3 and continue execution from that point.
- Start over from Step 1.
- If you understand this material, skip ahead to Line 21.
- If you are 18 years of age or older, do Part 1 beginning with Step 9; otherwise, do Part 2 beginning with Step 40.

...of unambiguous and effectively computable operations...

Algorithms are composed of things called "operations," but what do those operations look like? What types of building blocks can be used to construct an algorithm? The answer to these questions is that the operations used in an algorithm must meet two criteria—they must be *unambiguous*, and they must be *effectively computable*.

Here is a possible "algorithm" for making a cherry pie:

Step 1: Make the crust

Step 2: Make the cherry filling

Step 3: Pour the filling into the crust

Step 4: Bake at 350°F for 45 minutes

For a professional baker, this algorithm would be fine. He or she would understand how to carry out each of the operations listed. Novice cooks, like most of us, would probably understand the meaning of Steps 3 and 4. However, we would probably look at Steps 1 and 2, throw up our hands in confusion, and ask for clarification. We might then be given more detailed instructions.

Step 1: Make the crust
1.1 Take one and one-third cups flour
1.2 Sift the flour

1.3 Mix the sifted flour with one-half cup butter and one-fourth cup water
1.4 Roll into two 9-inch pie crusts

Step 2: Make the cherry filling
2.1 Open a 16-ounce can of cherry pie filling and pour into bowl
2.2 Add a dash of cinnamon and nutmeg, and stir

With this additional information, most people—even inexperienced cooks—would understand what to do and could successfully carry out this baking algorithm. However, there might be some people, perhaps young children, who still do not fully understand each and every line. For those people, we must go through the simplification process again and describe the ambiguous steps in even more elementary terms.

For example, the computing agent executing the algorithm might not know the meaning of the instruction "Sift the flour" in Step 1.2, and we would have to explain it further.

1.2 Sift the flour
1.2.1 Get out the sifter, which is the device shown on page A-9 of your cookbook, and place it directly on top of a 2-quart bowl
1.2.2 Pour the flour into the top of the sifter and turn the crank in a counterclockwise direction
1.2.3 Let all the flour fall through the sifter into the bowl

Now, even a child should be able to carry out these operations. But if that were not the case, then we would go through the simplification process yet one more time, until every operation, every sentence, every word was clearly understood.

An **unambiguous operation** is one that can be understood and carried out directly by the computing agent without further simplification or explanation. When an operation is unambiguous, we call it a *primitive operation,* or simply a **primitive** of the computing agent carrying out the algorithm. An algorithm must be composed entirely of primitives. Naturally, the primitive operations of different individuals (or machines) vary depending on their sophistication, experience, and intelligence, as is the case with the cherry pie recipe, which varies with the baking experience of the person following the instructions. Hence, an algorithm for one computing agent might not be an algorithm for another.

One of the most important questions we will answer in this text is, *What are the primitive operations of a typical modern computer system?* Which operations can a hardware processor "understand" in the sense of being able to carry out directly, and which operations must be further refined and simplified?

However, it is not enough for an operation to be understandable. It must also be *doable* by the computing agent. If an algorithm tells me to flap my arms really quickly and fly, I understand perfectly well what it is asking me to do. However, I am incapable of doing it. "Doable" means there exists a computational process that allows the computing agent to complete that operation successfully. The formal term for "doable" is **effectively computable**.

For example, the following is an incorrect technique for finding and printing the 100th prime number. (A prime number is a whole number not evenly divisible by any numbers other than 1 and itself, such as 2, 3, 5, 7, 11, 13, ...)

Step 1: Generate a list L of all the prime numbers: L_1, L_2, L_3, ...

Step 2: Sort the list L into ascending order

Step 3: Print out the 100th element in the list, L_{100}

Step 4: Stop

The problem with these instructions is in Step 1, "Generate a list L of *all* the prime numbers...." That operation cannot be completed. There are an infinite number of prime numbers, and it is not possible in a finite amount of time to generate the desired list L. No such computational process exists, and the operation described in Step 1 is not effectively computable. Here are some other examples of operations that, under certain circumstances, may not be effectively computable:

Set *number* to 0.
Set *average* to *(sum ÷ number)*. (Division by 0 is not permitted.)

Set N to -1.
Set the value of *result* to $\sqrt{N}$. (You cannot take the square root of negative values using real numbers.)

Add 1 to the current value of *x*. (What if *x* currently has no value?)

This last example explains why we had to initialize the value of the variable called *carry* to 0 in Step 1 of Figure 1.2. In Step 4, the algorithm says, "Add the two digits a_i and b_i to the current value of *carry* to get c_i." If *carry* has no current value, then when the computing agent tries to perform the instruction in Step 4, it will not know what to do, and this operation is not effectively computable.

...that produces a result...

Algorithms solve problems. To know whether a solution is correct, an algorithm must produce a result that is observable to a user, such as a numerical answer, a new object, or a change to its environment. Without some observable result, we would not be able to say whether the algorithm is right or wrong or even if it has completed its computations. In the case of the DVR algorithm (Figure 1.1), the result will be a set of recorded TV programs. The addition algorithm (Figure 1.2) produces an m-digit sum.

Note that we use the word *result* rather than *answer.* Sometimes it is not possible for an algorithm to produce the correct answer because for a given set of input, a correct answer does not exist. In those cases, the algorithm may produce something else, such as an error message, a red warning light, or an approximation to the correct answer. Error messages, lights, and approximations, although not necessarily what we wanted, are all observable results.

...and halts in a finite amount of time.

Another important characteristic of algorithms is that the result must be produced after the execution of a finite number of operations, and we must guarantee that the algorithm eventually reaches a statement that says, "Stop, you are done" or something equivalent. We have already pointed out that the shampooing algorithm was not well ordered because we did not know which statements to repeat in Step 4. However, even if we knew which block of statements to repeat, the algorithm would still be incorrect because it makes no provision to terminate. It will essentially run forever, or until we run out of hot water, soap, or patience. This is called an **infinite loop**, and it is a common error in the design of algorithms.

Figure 1.3 shows an algorithmic solution to the shampooing problem that meets all the criteria discussed in this section if we assume that you want to wash your hair twice. The algorithm of Figure 1.3 is well ordered. Each step is numbered, and the execution of the algorithm unfolds sequentially, beginning at Step 1 and proceeding from instruction i to instruction $i + 1$, unless the operation specifies otherwise. (For example, the iterative instruction in Step 3 says that after completing Step 6, you should go back and start again at Step 4 until the value of *WashCount* equals 2.) The intent of each operation is (we assume) clear, unambiguous, and doable by the person washing his or her hair. Finally, the algorithm will halt. This is confirmed by observing that *WashCount* is initially set to 0 in Step 2. Step 6 says to add 1 to *WashCount* each time we lather and rinse our hair, so it will take on the values 0, 1, 2, ... However, the iterative statement in Step 3 says stop lathering and rinsing when the value of *WashCount* reaches 2. At that point, the algorithm goes to Step 7 and terminates execution with the desired result: clean hair. (Although it is correct, do not expect to see this algorithm on the back of a shampoo bottle in the near future.)

As is true for any recipe or set of instructions, there is always more than a single way to write a correct solution. For example, the algorithm of Figure 1.3 could also be written as shown in Figure 1.4. Both of these are correct solutions to the shampooing problem. (Although they are both

FIGURE 1.3

Step	Operation
1	Wet your hair
2	Set the value of *WashCount* to 0
3	Repeat Steps 4 through 6 until the value of *WashCount* equals 2
4	Lather your hair
5	Rinse your hair
6	Add 1 to the value of *WashCount*
7	Stop, you have finished shampooing your hair

A correct solution to the shampooing problem

FIGURE 1.4

Step	Operation
1	Wet your hair
2	Lather your hair
3	Rinse your hair
4	Lather your hair
5	Rinse your hair
6	Stop, you have finished shampooing your hair

Another correct solution to the shampooing problem

correct, they are not necessarily equally elegant. This point is addressed in Exercise 9 at the end of the chapter.)

1.3.2 The Importance of Algorithmic Problem Solving

The instruction sequences in Figures 1.1–1.4 are examples of the types of algorithmic solutions designed, analyzed, implemented, and tested by computer scientists, although they are much shorter and simpler. The operations shown in these figures could be encoded into some appropriate language and given to a computing agent (such as a personal computer or a robot) to execute. The device would mechanically follow these instructions and successfully complete the task. The device could do this without having to understand the creative processes that went into the discovery of the solution and without knowing the principles and concepts that underlie the problem. The robot simply follows the steps in the specified order (a required characteristic of algorithms), successfully completing each operation (another required characteristic), and ultimately producing the desired result after a finite amount of time (also required).

Just as the Industrial Revolution of the nineteenth century allowed machines to take over the drudgery of repetitive physical tasks, the "computer revolution" of the twentieth and twenty-first centuries has enabled us to implement algorithms that mechanize and automate the drudgery of repetitive mental tasks, such as adding long columns of numbers, finding one specific name or account number within a massive database, sorting student records by course number, and retrieving hotel or airline reservations from a file containing millions of pieces of data. This mechanization process offers the prospect of enormous increases in productivity. It also frees people to do those things that humans do much better than computers, such as creating new ideas, setting policy, doing high-level planning, and determining the significance of the results produced by a computer. Certainly, these operations are a much more effective use of that unique computing agent called the human brain.

Practice Problems

Get a copy of the instructions that describe how to do the following:

1. Register for classes at the beginning of the semester.
2. Use the online catalog to see what is available in the college library on a given subject.
3. Place an order for a product on Amazon.
4. Do an "Advanced Search" using Google.
5. Add someone as a friend to your Facebook account.

Look over the instructions and decide whether they meet the definition of an algorithm given in this section. If not, explain why, and rewrite each set of instructions so that it constitutes a valid algorithm. Also state whether each instruction is a sequential, a conditional, or an iterative operation.

1.4 A Brief History of Computing

Although computer science is not simply a study of computers, there is no doubt that the field was formed and grew in popularity as a direct response to their creation and widespread use. This section takes a brief look at the historical development of computer systems.

The appearance of some technologies, such as the telephone, the light bulb, and the first heavier-than-air flight, can be traced directly to a single place, a specific individual, and an exact instant in time. Examples include the flight of Orville and Wilbur Wright on December 17, 1903, in Kitty Hawk, North Carolina, and the famous phrase "Mr. Watson–come here–I want to see you." uttered by Alexander Graham Bell over the first telephone on March 10, 1876.

Computers were not like that. They did not appear in a specific room on a given day as the creation of some individual genius. The ideas that led to the design of the first computers evolved over hundreds of years, with contributions coming from many people, each building on and extending the work of earlier discoverers.

1.4.1 The Early Period: Up to 1940

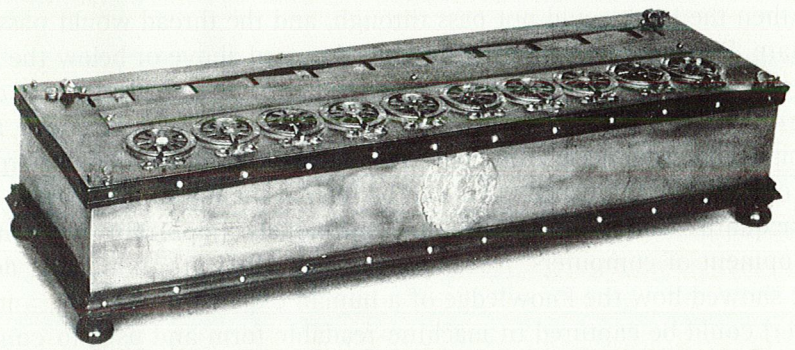
If this were a discussion of the history of mathematics and arithmetic instead of computer science, it would begin 3,000 years ago with the early work of the Greeks, Egyptians, Babylonians, Indians, Chinese, and Persians. All these

cultures were interested in and made important contributions to the fields of mathematics, logic, and numerical computation. For example, the Greeks developed the fields of geometry and logic; the Babylonians and Egyptians developed numerical methods for generating square roots, multiplication tables, and trigonometric tables used by early sailors; Indian mathematicians developed both the base-10 decimal numbering system and the concept of zero; and in the ninth century, the Persians developed algorithmic problem solving (as you learned in the Abu Ja'far Muhammad ibn Musa Al-Khwarizmi Special Interest Box earlier in the chapter).

The first half of the seventeenth century saw a number of important developments related to automating and simplifying the drudgery of arithmetic computation. (The motivation for this work appears to be the sudden increase in scientific research during the sixteenth and seventeenth centuries in the areas of astronomy, chemistry, and medicine. This work required the solution of larger and more complex mathematical problems.) In 1614, the Scotsman John Napier invented *logarithms* as a way to simplify difficult mathematical computations. The early seventeenth century also witnessed the development of new and quite powerful mechanical devices designed to help reduce the burden of arithmetic. The first *slide rule* appeared around 1622. In 1642, the French philosopher and mathematician Blaise Pascal designed and built one of the first *mechanical calculators* (named the *Pascaline*) that could do addition and subtraction. A model of this early calculating device is shown in Figure 1.5.

The famous German mathematician Gottfried Leibnitz (who, along with Isaac Newton, was one of the inventors of calculus) was also excited by the idea of automatic computation. He studied the work of Pascal and others, and in 1673, he constructed a mechanical calculator called *Leibnitz's Wheel* that could do not only addition and subtraction but multiplication and division as well. Both Pascal's and Leibnitz's machines used interlocking

FIGURE 1.5



The Pascaline, one of the earliest mechanical calculators

Source: INTERFOTO / Alamy Stock Photo

mechanical cogs and gears to store numbers and perform basic arithmetic operations. Considering the state of technology available to Pascal, Leibnitz, and others in the seventeenth century, these first calculating machines truly were mechanical wonders.

These early developments in mathematics and arithmetic were important milestones because they demonstrated how mechanization could simplify and speed up numerical computation. For example, Leibnitz's Wheel enabled seventeenth-century mathematicians to generate tables of mathematical functions many times faster than was possible by hand. (It is hard to believe in our modern high-tech society, but in the seventeenth century the generation of a table of logarithms could represent a *lifetime's* effort of one person!) However, the slide rule and mechanical calculators of Pascal and Leibnitz, although certainly impressive devices, were not computers. Specifically, they lacked two fundamental characteristics:

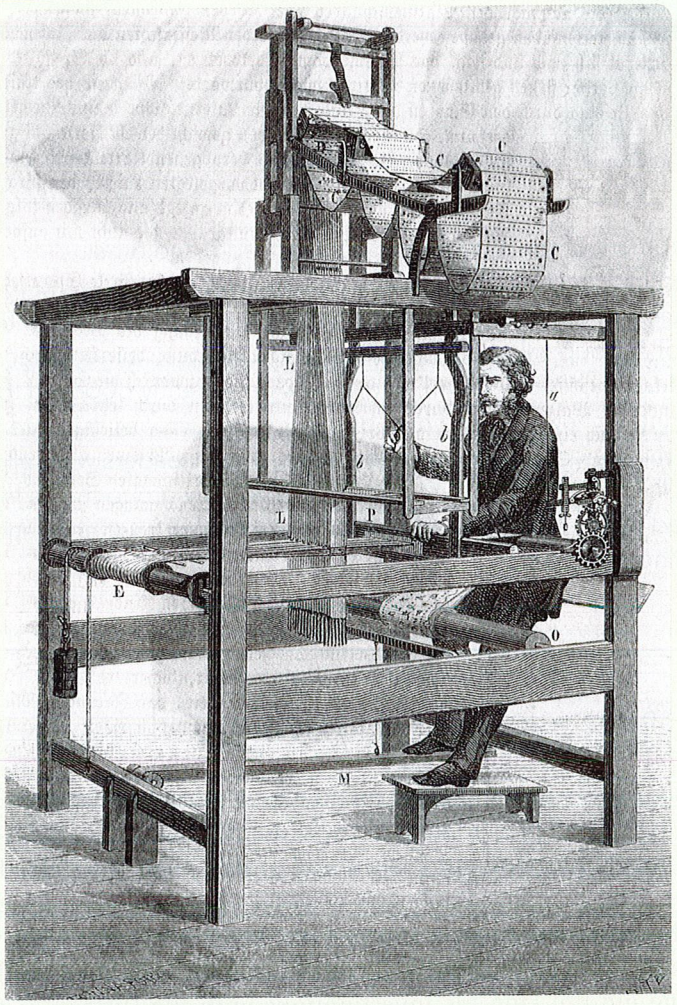
- They did not have a *memory* where information could be stored in machine-readable form.
- They were not *programmable.* A person could not provide *in advance* a sequence of instructions that could be executed by the device without manual intervention.

Surprisingly, the first actual "computing device" to include both these features was not created for the purposes of mathematical computations. Rather, it was a loom used for the manufacture of rugs and clothing. It was developed in 1801 by the Frenchman Joseph Jacquard. Jacquard wanted to automate the weaving process, at the time a painfully slow and cumbersome task in which each separate row of the pattern had to be set up by the weaver and an apprentice. Because of this, anything but the most basic style of clothing was beyond the means of most people.

Jacquard designed an automated loom that used *punched cards* to create the desired pattern (Figure 1.6). If there was a hole in the card in a particular location, then a hook could pass through the card, grasp a warp thread, and raise it to allow a second thread to pass underneath. If there was no hole in the card, then the hook could not pass through, and the thread would pass over the warp. Depending on whether the thread passed above or below the warp, a specific design was created. Each punched card described one row of the pattern. Jacquard connected the cards and fed them through his loom, and it automatically sequenced from card to card, weaving the desired pattern. The rows of connected punched cards can be seen at the top of the device.

Jacquard's loom represented an enormously important stage in the development of computers. Not only was it the first programmable device, but it showed how the knowledge of a human expert (in this case, a master weaver) could be captured in machine-readable form and used to control a machine that accomplished the same task automatically. Once the program was created, the expert was no longer needed. The lowliest apprentice could load the cards into the loom, turn it on, and produce a finished, high-quality product over and over again.

FIGURE 1.6



Drawing of the Jacquard loom

Source: Bettmann/Getty Images

These pioneers had enormous influence on the designers and inventors who came after them, among them a mathematics professor at Cambridge University named Charles Babbage. Babbage was interested in automatic computation. In 1823, he extended the ideas of Pascal and Leibnitz and constructed a working model of the largest and most sophisticated mechanical calculator of its time. This machine, called the *Difference Engine,* could do addition, subtraction, multiplication, and division to six significant digits, and it could solve polynomial equations and other complex mathematical problems as well. Babbage tried to construct a larger model of the Difference Engine that would be capable of working to an accuracy of 20 significant

The Original "Technophobia"

The development of the automated Jacquard loom and other technological advances in the weaving industry was so frightening to the craft guilds of the early nineteenth century that in 1811 it led to the formation of a group called the Luddites. The Luddites, named after their leader Ned Ludd of Nottingham, England, were violently opposed to this new manufacturing technology, and they burned down factories that attempted to use it. The movement lasted only a few years and its leaders were all jailed, but their name lives on today as a pejorative term for any group that is frightened and angered by the latest developments in any branch of science and technology, including computers.

digits, but after 12 years of work he had to give up his quest. The technology available in the 1820s and 1830s was not sufficiently advanced to manufacture cogs and gears to the precise tolerances his design required. Like Leonardo da Vinci's helicopter or Jules Verne's atomic submarine, Babbage's ideas were fundamentally sound but years ahead of their time. (In 1991, the London Museum of Science, using Babbage's original plans, built an actual working model of the Difference Engine. It was 7 feet high and 11 feet wide, weighed 5 tons, and had 4,000 moving parts. It worked exactly as Babbage had planned.)

Babbage did not stop his investigations with the Difference Engine. In the 1830s, he designed a more powerful and general-purpose computational machine that could be configured to solve a much wider range of numerical problems. His machine had four basic components: a *mill* to perform the arithmetic manipulation of data, a *store* to hold the data, an *operator* to process the instructions contained on punched cards, and an *output unit* to put the results onto separate punched cards. Although it would be about 110 years before a "real" computer would be built, Babbage's proposed machine, called the Analytical Engine, is amazingly similar in design to a modern computer. The four components of the Analytical Engine are virtually identical in function to the four major components of today's computer systems:

Babbage's Term	*Modern Terminology*
mill	arithmetic/logic unit
store	memory
operator	processor
output unit	input/output

Babbage died before a working steam-powered model of his Analytical Engine could be completed, but his ideas lived on to influence others, and many computer scientists consider the Analytical Engine the first "true" computer system, even if it existed only on paper and in Babbage's dreams.

Another person influenced by the work of Pascal, Jacquard, and Babbage was a young statistician at the U.S. Census Bureau named Herman Hollerith. Because of the rapid increase in immigration to America at the end of the nineteenth century, officials estimated that doing the 1890 enumeration manually would take from 10 to 12 years. The 1900 census would begin before the previous one was finished. Something had to be done.

Hollerith designed and built programmable card-processing machines that could automatically read, tally, and sort data entered on punched cards. Census data were coded onto cards using a machine called a *keypunch*. The cards were taken either to a *tabulator* for counting and tallying or to a *sorter* for ordering alphabetically or numerically. Both of these machines were programmable (via wires and plugs) so that the user could specify such things as which card columns should be tallied and in what order the cards should be sorted. In addition, the machines had a small amount of memory to store results. Thus, they had all four components of Babbage's Analytical Engine.

Hollerith's machines were enormously successful, and they were one of the first examples of the use of automated information processing to solve large-scale, real-world problems. Whereas the 1880 census required 8 years to be completed, the 1890 census was finished in about 1 year, even though there was a 26% increase in the U.S. population during that decade.

These machines were not really general-purpose computers because each machine could do only a single task such as tabulate or sort. Nevertheless, Hollerith's card machines were a very clear and very successful demonstration of the enormous advantages of automated information processing. This fact was not lost on Hollerith, who left the Census Bureau in 1902 to run his own Tabulating Machine Company to build and sell these machines. He planned to market his new product to a country that was just entering the Industrial Revolution and that, like the Census Bureau, would be generating and processing enormous volumes of inventory, production, accounting, and sales data. His punched-card machines became the dominant form of data-processing equipment during the first half of the twentieth century, well into the 1950s and 1960s. During this period, virtually every major U.S. corporation had data-processing rooms filled with keypunches, sorters, and tabulators, as well as drawer upon drawer of punched cards. In 1924, Hollerith's company changed its name to IBM, and it eventually evolved into the largest computing company in the world.

We have come a long way from the 1640s and the Pascaline, the early adding machine constructed by Pascal. We have seen the development of more powerful mechanical calculators (Leibnitz), automated programmable manufacturing devices (Jacquard), a design for the first computing device (Babbage), and the initial applications of information processing on a massive scale (Hollerith). However, we still have not yet entered the "computer age." That did not happen until around 1940, and it was motivated by an

Charles Babbage (1791–1871) Ada Augusta Byron, Countess of Lovelace (1815–1852)

Charles Babbage, the son of a banker, was born into a life of wealth and comfort in eighteenth-century England. He attended Cambridge University and displayed an aptitude for mathematics and science. He was also an inventor and "tinkerer" who loved to build all sorts of devices. Among the devices he constructed were unpickable locks, skeleton keys, speedometers, and even the first cow catcher for trains. His first and greatest love, though, was mathematics, and he spent much of his life creating machines to do automatic computation. Babbage was enormously impressed by the work of Jacquard in France. (In fact, Babbage had on the wall of his home a woven portrait of Jacquard that was created using 24,000 punched cards.) He spent the last 30–40 years of his life trying to build a computing device, the Analytical Engine, based on Jacquard's ideas.

In that quest, he was helped by Countess Ada Augusta Byron, daughter of the famous English poet, Lord Byron. The countess was introduced to Babbage and was enormously impressed by his ideas about the Analytical Engine. As she put it, "We may say most aptly that the Analytical Engine weaves algebraic patterns just as the Jacquard Loom weaves flowers and leaves." Lady Lovelace worked closely with Babbage to specify how to organize instructions for the Analytical Engine to solve a particular mathematical problem. Because of that pioneering work, she is generally regarded as history's first computer programmer.

Babbage died in 1871 without realizing his dream. His work was generally forgotten until the twentieth century, when it became instrumental in moving the world into the computer age.

event that, unfortunately, has fueled many of the important technological advances in human history—the outbreak of war.

1.4.2 The Birth of Computers: 1940–1950

World War II created another, quite different set of information-based problems. Instead of inventory, sales, and payroll, the concerns became ballistics tables, troop deployment data, and secret codes. A number of research projects were started, funded largely by the military, to build

automatic computing machines to perform these tasks and assist the Allies in the war effort.

Beginning in 1937, the U.S. Navy and IBM jointly funded a project under the direction of Professor Howard Aiken at Harvard University to build a computing device called Mark I. This was a general-purpose, electromechanical programmable computer that used a mix of relays, magnets, and gears to process and store data. The Mark I was the first computing device to use the base-2 binary numbering system, which we will discuss in Chapter 4. It used electromechanical switches and electric current to represent the two binary values, off for 0, on for 1. Until then, computing machines had used decimal representation, typically using a 10-toothed gear, each tooth representing one of the digits from 0 to 9. The Mark I was completed in 1944, about 110 years after Babbage's dream of the Analytical Engine, and is generally considered one of the first working general-purpose computers. The Mark I had a memory capacity of 72 numbers, and it could be programmed to perform a 23-digit multiplication in the lightning-like time of 4 seconds. Although laughably slow by modern standards, the Mark I was operational for almost 15 years, and it carried out a good deal of important mathematical work for the U.S. during the war.

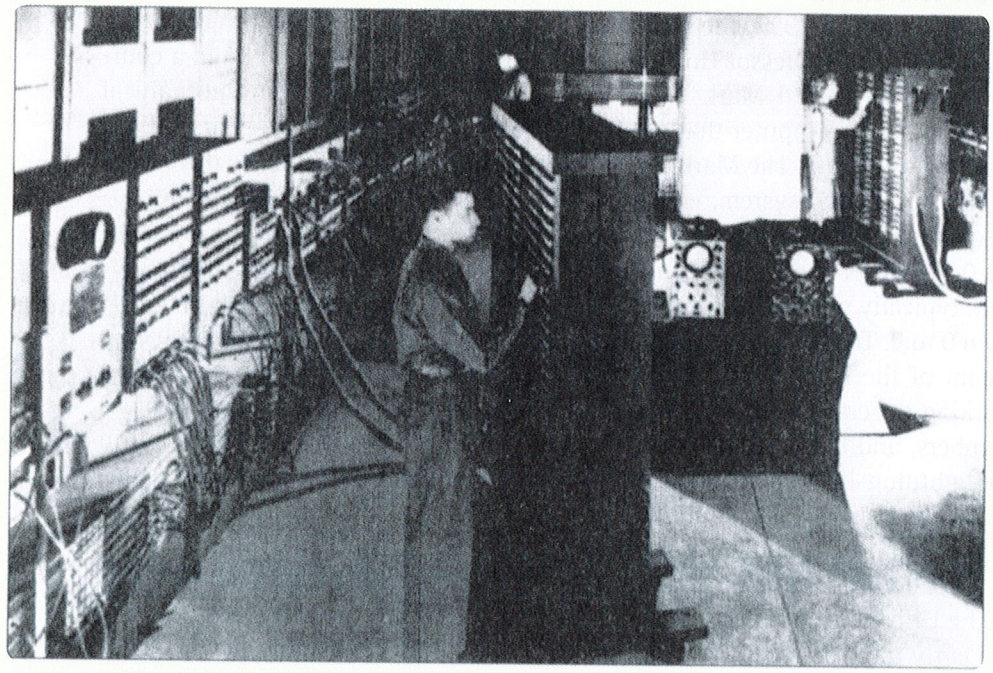
At about the same time, a much more powerful machine was taking shape at the University of Pennsylvania in conjunction with the U.S. Army. During the early days of World War II, the Army was producing many new artillery pieces, but it found that it could not produce the firing tables equally as fast. These tables told the gunner how to aim the gun on the basis of such input as distance to the target and current temperature, wind, and elevation. Because of the enormous number of variables and the complexity of the computations (which use both trigonometry and calculus), these firing tables were taking more time to construct than the gun itself—a skilled person with a desk calculator required about 20 hours to analyze a single 60-second trajectory.

To help solve this problem, in 1943 the Army initiated a research project with J. Presper Eckert and John Mauchly of the University of Pennsylvania to build a completely electronic computing device. The machine, dubbed the ENIAC (Electronic Numerical Integrator and Calculator), was completed in 1946 (too late to assist in the war effort) and was the first fully electronic general-purpose programmable computer. This pioneering machine is shown in Figure 1.7.

ENIAC contained 18,000 vacuum tubes and nearly filled a building; it was 100 feet long and 10 feet high and weighed 30 tons. Because it was fully electronic, it did not contain any of the slow mechanical components found in Mark I, and it executed instructions much more rapidly. The ENIAC could add two 10-digit numbers in about 1/5,000 of a second and could multiply two numbers in 1/300 of a second, 1,000 times faster than the Mark I.

The Mark I and ENIAC are two well-known examples of early computers, but they are by no means the only ones of that era. For example, the ABC system (Atanasoff–Berry Computer), designed and built by Professor John Atanasoff and his graduate student Clifford Berry at Iowa State University, was actually the first electronic computer, constructed during the period 1939–1942. However, it never received equal recognition because it was useful for only one task, solving systems of simultaneous linear equations.

FIGURE 1.7



Photograph of the ENIAC computer

Source: U.S. Army, from the Collections of the University of Pennsylvania Archives

In England, a computer called Colossus was built in 1943 under the direction of Alan Turing, a famous mathematician and computer scientist whom we will meet again in Chapter 12. This machine, one of the first computers built outside the United States, was used to crack the famous German Enigma code that the Nazis believed to be unbreakable. Colossus has also not received as much recognition as ENIAC because of the secrecy that shrouded the Enigma project. Its very existence was not widely known until the late 1970s, more than 30 years after the end of World War II.

At about the same time that Colossus was taking form in England, a German engineer named Konrad Zuse was working on a computing device for the German army. The machine, code named Z1, was similar in design to the ENIAC—a programmable, general-purpose, fully electronic computing device. Fortunately for the Allied forces, the Z1 project was not completed before the end of World War II.

Although the machines just described—ABC, Mark I, ENIAC, Colossus, and Z1—were computers in the fullest sense of the word (they had memory and were programmable), they did not yet look like modern computer systems. One more step was necessary, and that step was taken in 1946 by the one individual who was most instrumental in creating the computer as we know it today, John Von Neumann.

Von Neumann was not only one of the most brilliant mathematicians who ever lived, he was also a genius in many other areas as well, including experimental physics, chemistry, economics, and computer science. Von Neumann, who taught at Princeton University, had worked with Eckert and Mauchly on the ENIAC project at the University of Pennsylvania. Even though that project was successful, he recognized a number of fundamental shortcomings in ENIAC. In 1946, he proposed a radically different computer design based on a model called the stored program computer. Until then, all computers were programmed *externally* using wires, connectors, and plugboards. The memory unit stored only data, not instructions. For each different problem, users had to rewire virtually the entire computer. For example, the plugboards on the ENIAC contained 6,000 separate switches, and reprogramming the ENIAC involved specifying the new settings for all these switches—not a trivial task.

Von Neumann proposed that the instructions that control the operation of the computer be encoded as binary values and stored internally in the memory unit along with the data. To solve a new problem, instead of rewiring the machine, you would rewrite the sequence of instructions—that is, create a new program. Von Neumann invented programming as it is known today.

The model of computing proposed by Von Neumann included many other important features found on all modern computing systems, and to honor him this model of computation has come to be known as the Von Neumann architecture. We will study this architecture in great detail in Chapters 4 and 5.

Von Neumann's research group at the University of Pennsylvania implemented his ideas, and they built one of the first stored program computers, called EDVAC, in 1949. At about the same time, a stored program computer called EDSAC was built at Cambridge University in England under the direction of Professor Maurice Wilkes. The appearance of these machines and others like them ushered in the modern computer age. Even though they were much slower, bulkier, and less powerful than our current machines, EDVAC and EDSAC executed programs in a fashion surprisingly similar to the miniaturized and immensely more powerful computers of the twenty-first century. A commercial model of the EDVAC, called UNIVAC I—the first computer actually sold—was built by Eckert and Mauchly and delivered to the U.S. Bureau of the Census on March 31, 1951. (It ran for 12 years before it was retired, shut off for the last time, and moved to the Smithsonian Institution.) This date marks the true beginning of the "computer age."

The importance of Von Neumann's contributions to computer systems development cannot be overstated. Although his original proposals are about 70 years old, virtually every computer built today is a Von Neumann machine in its basic design. A lot has changed in computing, and a sleek new iPad and the bulky EDVAC would appear to have little in common. However, the basic principles on which these two very disparate machines are constructed are virtually identical, and the same theoretical model underlies their operation. There is an old saying in computer science: "There is nothing new since Von Neumann!" This saying is certainly not true (much *has* happened), but it demonstrates the importance and amazing staying power of Von Neumann's original design.

John Von Neumann (1903–1957)

John Von Neumann was born in Budapest, Hungary. He was a child prodigy who could divide 8-digit numbers in his head by the age of 6. He was a genius in virtually every field that he studied, including physics, economics, engineering, and mathematics. At 18, he received an award as the best mathematician in Hungary, a country known for excellence in the field, and he received his Ph.D., summa cum laude, at 21. He came to the United States in 1930 to be a guest lecturer at Princeton University and taught there for 3 years. Then, in 1933 he became one of the founding members (along with Albert Einstein) of the Institute for Advanced Studies, where he worked for 20 years.

Source: Los Alamos National Laboratory

He was one of the most brilliant minds of the twentieth century, a true genius in every sense, both good and bad. He could do prodigious mental feats in his head, and his thought processes usually raced far ahead of "ordinary" mortals, who found him quite difficult to work with. One of his colleagues described him as possessing the most fearsome technical intellect of the century. Another joked that "Johnny wasn't really human, but after living among them for so long, he learned to do a remarkably good imitation of one."

Von Neumann was a brilliant theoretician who did pioneering work in pure mathematics, operations research, game theory, and theoretical physics. He was also an engineer, concerned about practicalities and real-world problems, and it was this interest in applied issues that led Von Neumann to design and construct the first stored program computer. One of the early computers built by the RAND Corp. in 1953 was affectionately called "Johnniac" in his honor, although Von Neumann detested that name. Following its shutdown, it was moved to the Computer History Museum in Mountain View, California.

1.4.3 The Modern Era: 1950 to the Present

The last 65 or so years of computer development have involved taking the Von Neumann architecture and improving it in terms of hardware and software. Since 1950, computer systems development has been primarily an *evolutionary* process, not a revolutionary one. The enormous number of changes in computers in recent decades has made them faster, smaller, cheaper, more reliable, and easier to use but has not drastically altered their basic underlying structure.

The period 1950–1957 (these dates are very rough approximations) is often called the *first generation* of computing. This era saw the appearance of UNIVAC I, the first computer built for sale, and the IBM 701, the first computer

And the Verdict Is . . .

Our discussion of what was happening in computing from 1939 to 1946 showed that many groups were involved in designing and building the first computers. Therefore, it would seem that no single individual can be credited with the title "Inventor of the Electronic Digital Computer."

Surprisingly, that is not true. In February 1964, the Sperry Rand Corp. (now UNISYS) was granted a U.S. patent on the ENIAC computer as the first fully electronic computing device, with J. Presper Eckert and John Mauchly listed as its designers and builders. However, in 1967 a suit was filed in U.S. District Court in Minneapolis, Minnesota, to overturn that patent. The suit, *Honeywell v. Sperry Rand*, was heard before U.S. Federal Judge Earl Larson, and on October 19, 1973, Judge Larson handed down his verdict. (This enormously important verdict was never given the media coverage it deserved because it happened in the middle of the Watergate hearings and on the very day that Vice President Spiro Agnew resigned in disgrace for tax fraud.) Judge Larson overturned the ENIAC patent on the basis that Eckert and Mauchly had been significantly influenced in their 1943–1944 work on ENIAC by earlier research and development work by John Atanasoff at Iowa State University. During the period 1939–1943, Mauchly had communicated extensively with Atanasoff and had even traveled to Iowa to see the ABC machine in person. In a sense, the verdict declared that Professor Atanasoff was the inventor of the electronic digital computer. This decision was never appealed. Therefore, the official honor of having designed and built the first computer, at least in U.S. District Court, goes to Professor John Vincent Atanasoff.

On November 13, 1990, in a formal ceremony at the White House, President George H.W. Bush awarded Professor Atanasoff the National Medal of Technology for his pioneering contributions to the development of the computer.

built by the company that would soon become a leader in this new field. These early systems were similar in design to EDVAC, and they were bulky, expensive, slow, and unreliable. They used vacuum tubes for processing and storage, and they were extremely difficult to maintain. The simple act of turning on the machine could blow out a dozen tubes! For this reason, first-generation machines were used only by trained personnel and only in specialized locations such as large corporations, government and university research labs, and military installations, which could provide this expensive support environment.

The *second generation* of computing, roughly 1957–1965, heralded a major change in the size and complexity of computers. In the late 1950s, the bulky vacuum tube was replaced by a single transistor only a few millimeters in size, and memory was now constructed using tiny magnetic cores only 1/50th of an inch in diameter. (We will introduce and describe both devices in Chapter 4.) These technologies not only dramatically reduced the size of computers but also increased their reliability and reduced costs. Suddenly, buying and using a computer became a real possibility for some small and medium-sized businesses, colleges, and government agencies. This was also the era of the appearance of FORTRAN and COBOL, the first high-level (English-like) programming languages. (We will study this type of programming language in Chapters 9 and 10.) Now it was no longer necessary to be an electrical engineer to solve a problem on a computer. One simply needed to learn how to write commands in a high-level language. The occupation called *programmer* was born.

This miniaturization process continued into the *third generation* of computing, which lasted from about 1965 to 1975. This was the era of the *integrated circuit*. Rather than using discrete electronic components, integrated circuits with transistors, resistors, and capacitors were photographically etched onto a piece of silicon, which further reduced the size and cost of computers. From building-sized to room-sized, computers now became desk-sized, and this period saw the birth of the first minicomputer—the PDP-1 manufactured by the Digital Equipment Corp. It also saw the birth of the *software industry,* as companies sprang up to provide programs such as accounting packages and statistical programs to the ever-increasing numbers of computer users. By the mid-1970s, computers were no longer a rarity. They were being widely used throughout industry, government, the military, and education.

The *fourth generation,* roughly 1975–1985, saw the appearance of the first microcomputer. Integrated circuit technology had advanced to the point that a complete computer system could be contained on a single circuit board that you could hold in your hand. The desk-sized machine of the early 1970s now became a desktop machine, shrinking to the size of a typewriter. The Altair 8800, the world's first microcomputer, appeared in January 1975 (see the Special Interest Box on the next page).

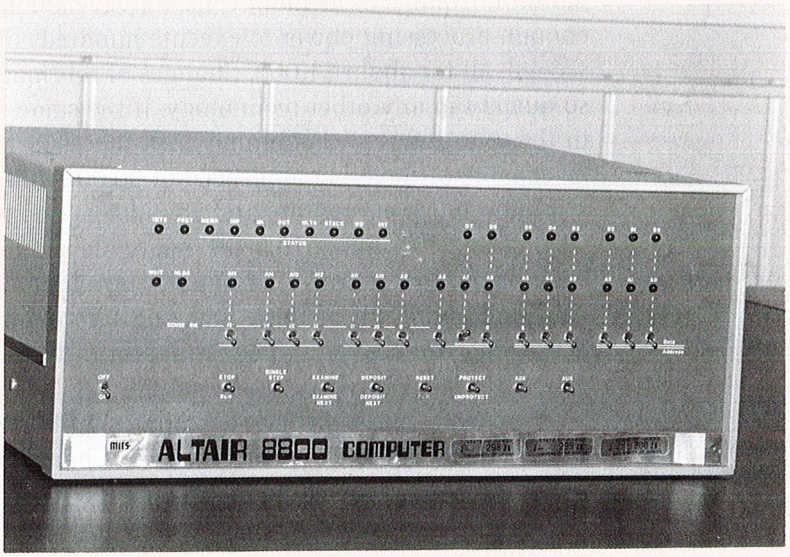
It soon became unusual *not* to see a computer on someone's desk. The software industry poured forth all types of new packages—spreadsheets, databases, word processors, and presentation graphics—to meet the needs of the burgeoning user population. This era saw the appearance of the first *computer networks*, as users realized that much of the power of computers lies in their facilitation of communication with other users. (We will look at networking in great detail in Chapter 7.) *Electronic mail* became an important application. Because so many users were computer novices, the concept of *user-friendly systems* emerged. This included new *graphical user interfaces* with pull-down menus, icons, and other visual aids to make computing easier and more fun. *Embedded systems*—devices that contain a computer to control their internal operation—first appeared during this generation. Computers were becoming small enough to be placed inside cars, thermostats, microwave ovens, and wristwatches.

The World's First Microcomputer

The Altair 8800, shown below, was the first microcomputer and made its debut on the cover of *Popular Electronics* in January 1975. Its developer, Ed Roberts, owned a tiny electronics store in Albuquerque, New Mexico. His company was in desperate financial shape when he read about a new microprocessor from Intel, the Intel 8080. Roberts reasoned that this new chip could be used to sell a complete personal computer in kit form. He bought these new chips from Intel at the bargain basement price of $75 each and packaged them in a kit called the Altair 8800 (named after a location in the TV series Star Trek), which he offered to hobbyists for $397. Roberts figured he might sell a few hundred kits a year, enough to keep his company afloat temporarily. He ended up selling hundreds of them per day! The Altair microcomputer kits were so popular that he could not keep them in stock, and legend has it that people even drove to New Mexico and camped out in the parking lot to buy their computers.

This is particularly amazing in view of the fact that the original Altair was difficult to assemble and had only 256 memory cells, no I/O devices, and no software support. To program it, the user had to enter binary machine language instructions directly from the console switches. But even though it could do very little, people loved it because it was a real computer, and it was theirs.

The Intel 8080 chip did have the capability of running programs written in the language called BASIC that had been developed at Dartmouth in the early 1960s. A small software company located in Washington state wrote Ed Roberts a letter telling him that it had a BASIC compiler that could run on his Altair, making it much easier to use. That company was called Microsoft—and, as they say, the rest is history.



Source: University of Hawai'i at Hilo Graphics Services

The *fifth generation*, 1985–?, is where we are today. However, so much is changing so fast that the concept of distinct generations of computer development has outlived its usefulness. In computer science, change is now a constant companion. Some of the recent developments in computer systems include the following:

- Massively parallel processors containing millions of processors and capable of quadrillions (10^{15}) of computations per second
- Ultra-high-resolution graphics for 3D imaging, animation, movie making, video games, and virtual reality
- Powerful multimedia user interfaces incorporating sound, voice recognition, touch, photography, video, television, and body measurement data such as fingerprints, heartbeat, and retinal scans
- Integrated digital devices incorporating data, television, telephone, camera, fax, the Internet, and medical monitoring
- Self-driving, computer-controlled cars and automated computerized robotics
- Massive cloud storage devices capable of holding 100 exabytes (10^{20}) of data; that is equivalent to 100 billion gigabytes
- Ubiquitous computing, in which miniature computers are embedded into cars, cameras, kitchen appliances, home heating systems, clothing, and even our bodies

In only a few decades, computers have progressed from the UNIVAC I, which cost millions of dollars, had a few thousand memory locations, and was capable of only a few thousand operations per second, to today's top-of-the-line graphics design workstations with a high-definition, flat panel monitor, trillions of memory cells, massive amounts of external storage, and enough processing power to execute hundreds of billions of instructions per second, all for about $1,000. Changes of this magnitude have never occurred so quickly in any other technology. If the same rate of change had occurred in the auto industry, beginning with the 1909 Model-T, today's cars would be capable of traveling at a speed of 20,000 miles per hour, would get about 1 million miles per gallon, and would cost about $1.00!

Figure 1.8 summarizes the major developments that occurred during each of the generations of computer development discussed in this section. And underlying all of these amazing improvements, the theoretical model describing the design and construction of computers has not changed significantly in the last 70 years.

However, many people feel that significant and important structural changes are on the way. At the end of Chapter 5, we will introduce models of computing that are fundamentally quite different from the Von Neumann architecture in use today. These totally new approaches (e.g., quantum computing) may be the models used in the twenty-second century and beyond.

FIGURE 1.8

Generation	Approximate Dates	Major Advances
First	1950–1957	First commercial computers First symbolic programming languages Use of binary arithmetic, vacuum tubes for storage Punched card input/output
Second	1957–1965	Transistors and core memories First disks for mass storage Size reduction, increased reliability, lower costs First high-level programming languages First operating systems
Third	1965–1975	Integrated circuits Further reduction in size and cost, increased reliability First minicomputers Time-shared operating systems Appearance of the software industry First set of computing standards for compatibility between systems
Fourth	1975–1985	Large-scale and very-large-scale integrated circuits Further reduction in size and cost, increased reliability First microcomputers Growth of new types of software and of the software industry Computer networks Graphical user interfaces
Fifth	1985–?	Ultra-large-scale integrated circuits Supercomputers and parallel processors Laptops, tablets, smartphones, and handheld wireless devices Mobile computing Massive external data storage devices Ubiquitous computing High-resolution graphics, visualization, virtual reality Worldwide networks and cloud computing Multimedia user interfaces Widespread use of digitized sound, images, and movies

Some of the major advancements in computing

DEFINITION

Computer science: the study of algorithms, including

1. Their formal and mathematical properties
2. Their hardware realizations
3. Their linguistic realizations
4. Their applications

1.5 Organization of the Text

This book is divided into six separate sections, called levels, each of which addresses one aspect of the definition of computer science that appears at the beginning of this chapter. Let's repeat the definition and see how it maps into the sequence of topics to be presented.

Computer science is the study of algorithms, including

1. *Their formal and mathematical properties.* Level 1 of the text (Chapters 2 and 3) is titled "The Algorithmic Foundations of Computer Science." It continues the discussion of algorithmic problem solving begun in Sections 1.2 and 1.3 by introducing important mathematical and logical properties of algorithms. Chapter 2 presents the development of several algorithms that solve important technical problems—certainly more "technical" than shampooing your hair. It also looks at concepts related to the problem-solving process, such as how we discover and create good algorithms, what notation we can use to express our solutions, and how we can check to see whether our proposed algorithm correctly solves the desired problem.

   Our brute force chess example illustrates that it is not enough simply to develop a correct algorithm; we also want a solution that is efficient and that produces the desired result in a reasonable amount of time. (Would you want to market a chess-playing program that takes 10^{25} years to make its first move?) Chapter 3 describes ways to compare the efficiency of different algorithms and select the best one to solve a given problem. The material in Level 1 provides the necessary foundation for a study of the discipline of computer science.

2. *Their hardware realizations.* Although our initial look at computer science investigated how an algorithm behaved when executed by some abstract "computing agent," we ultimately want to execute our algorithms on "real" machines to get "real" answers. Level 2 of the text (Chapters 4 and 5) is titled "The Hardware World," and it looks at how to design and construct computer systems. It approaches this topic from two quite different viewpoints.

   Chapter 4 presents a detailed discussion of the underlying hardware. It introduces the basic building blocks of computers—binary numbers, transistors, logic gates, and circuits—and shows how these elementary electronic devices can be used to construct components to perform arithmetic and logic functions such as addition, subtraction, comparison, and sequencing. Although it is both interesting and important, this perspective produces a rather low-level view of a computer system. It is difficult to understand how a computer works by studying only these elementary components, just as it would be difficult to understand human behavior by investigating the behavior of individual cells. Therefore, Chapter 5 takes a higher-level view of computer hardware. It looks at computers not as a bunch of wires and circuits but as an integrated

collection of subsystems called memory, processor, storage, input/output, and communications. It will explain in great detail the principles of the Von Neumann architecture introduced in Section 1.4.

A study of computer systems can be done at an even higher level. To understand how a computer works, we do not need to examine the functioning of every one of the thousands of components inside a machine. Instead, we need only be aware of a few critical pieces that are essential to our work. From the user's perspective, everything else is superfluous. This "user-oriented" view of a computer system and its resources is called a **virtual machine** or a **virtual environment**. A virtual machine is composed only of the resources that the user perceives rather than of all the hardware resources that actually exist.

This viewpoint is analogous to our level of understanding of what happens under the hood of a car. There may be thousands of mechanical components inside an automobile engine, but most of us concern ourselves only with the items reported on the dashboard—for example, oil pressure, fuel level, engine temperature. This is our "virtual engine," and that is all we need or want to know. We are all too happy to leave the remaining details about engine design to our friendly neighborhood mechanic.

Level 3 (Chapters 6, 7, and 8), titled "The Virtual Machine," describes how a virtual environment is created using a component called *system software*. Chapter 6 takes a look at the most important and widely used piece of system software on a modern computer system, the *operating system*, which controls the overall operation of a computer and makes it easier for users to access. Chapter 7 then goes on to describe how this virtual environment can extend beyond the boundaries of a single system as it examines how to interconnect individual machines into *computer networks* and *distributed systems* that provide users with access to a huge collection of computer systems and information as well as an enormous number of other users. It is the system software, and the virtual machine it creates, that makes computer hardware manageable and usable. Finally, Chapter 8 discusses a critically important component of a virtual machine—the *security system* that validates who you are and ensures that you are not attempting to carry out an improper, illegal, or unsafe operation. As computers become central to the management of such sensitive data as medical records, military information, and financial data, this aspect of system software is taking on even greater importance.

3. *Their linguistic realizations.* After studying hardware design, computer organization, and virtual machines, you will have a good idea of the techniques used to design and build computers. In the next section of the text, we ask the question, how can this hardware be used to solve important and interesting problems? Level 4, titled "The Software World" (Chapters 9–12), takes a look at what is involved in designing and implementing computer software. It investigates the programs and instruction sequences executed by the hardware, rather than the hardware itself.

Chapter 9 compares several high-level programming languages and introduces fundamental concepts related to the topic of computer programming regardless of the particular language being studied. This single chapter is certainly not intended to make you a proficient programmer, and this book is not meant to be a programming text. Instead, its purpose is to illustrate some basic features of modern programming languages and give you an appreciation for the interesting and challenging task of the computer programmer. Rather than print a separate version of this text for each programming language, the textual material specific to each language can be found on the website for this text, and you can download the pages for the language specified by your instructor and used in your class. See the Preface of this text for instructions on accessing these webpages.

There are many programming languages, such as C++, Python, Java, and Perl, that can be used to encode algorithms. Chapter 10 provides an overview of a number of different languages and language models in current use, including the functional and parallel models. Chapter 11 describes how a program written in a high-level programming language can be translated into the low-level machine language codes first described in Chapter 5. Finally, Chapter 12 shows that, even when we marshal all the powerful hardware and software ideas described in the first 11 chapters, problems exist that cannot be solved algorithmically. Chapter 12 demonstrates that there are, indeed, limits to computing.

4. *Their applications.* Most people are concerned not with creating programs but with using programs, just as there are few automotive engineers but many, many drivers. Level 5, titled "Applications" (Chapters 13–16), moves from *how* to write a program to *what* these programs can do.

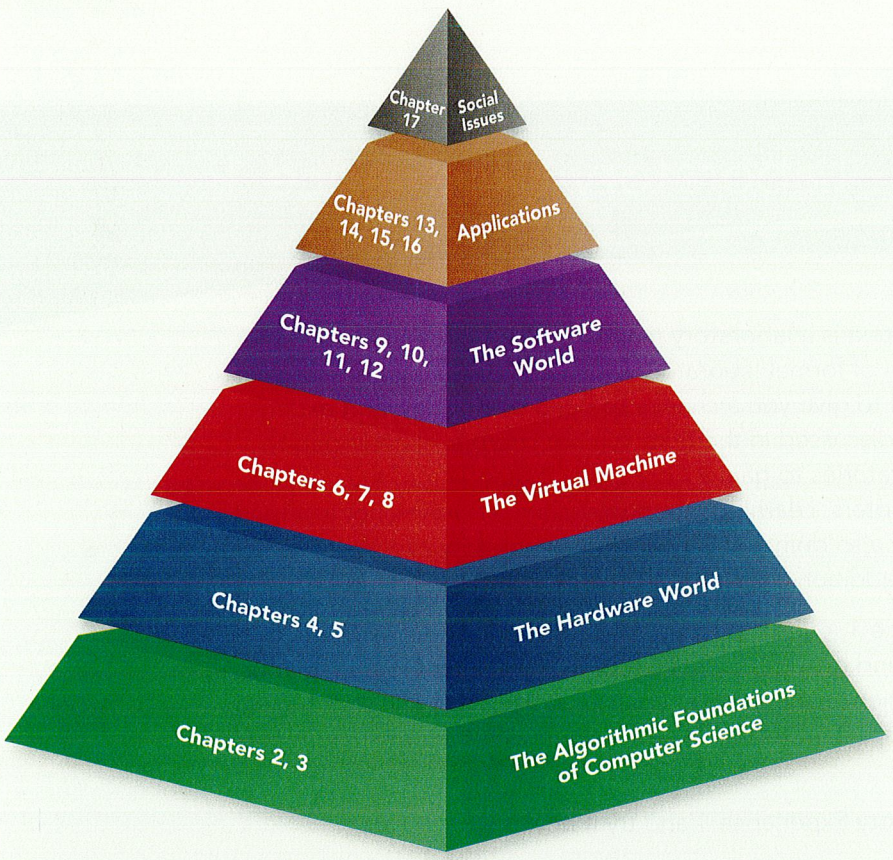
   Chapters 13 through 16 explore just a few of the many important and rapidly growing applications of computers, such as simulation, visualization, ecommerce, databases, artificial intelligence, computer graphics, and entertainment. Amazing as these applications seem, underneath their "shiny covers" they rely on the computer concepts of earlier chapters. Of course, we cannot possibly survey all the ways in which computers are being used today or will be used in the future. Indeed, there is hardly an area in our modern, complex society that has not been affected in some important way by information technology. (An excellent demonstration of this is that there are now 2.8 million apps available to Android users and 2.2 million available for downloading from the Apple App Store.) Readers interested in applications not discussed here should seek readings specific to their own areas of interest.

   Some computer science professionals are not concerned with building computers, creating programs, or using any of the applications just described. Instead, they are interested in the social and cultural impact—both positive and negative—of this ever-changing technology. The sixth level of this text addresses this important perspective on computer science. This is not part of the original definition of computer science but has become an important area of study. In Level 6, titled "Social Issues" (Chapter 17), we move to the highest level of abstraction—the view furthest removed from the computer itself—to discuss social, ethical, legal, and professional issues

related to computer and information technology. These issues are critically important because even individuals not directly involved in developing or using computers are deeply affected by them, just as society has been drastically and permanently altered by such technological developments as telephones, televisions, automobiles, and nuclear power. This last chapter takes a look at such thorny and difficult topics as computer crime, information privacy, and intellectual property. It also looks at one of the most important phenomena supported by this new technology, the creation of social networks such as Facebook, Twitter, LinkedIn, and Pinterest. Because it is impossible to resolve all the complex questions that arise in these areas, our intent is simply to raise your awareness and provide some decision-making tools to help you reach your own conclusions.

The overall six-layer hierarchy of this text is summarized in Figure 1.9. The organizational structure diagrammed in Figure 1.9 is one of the most

FIGURE 1.9



Organization of the text into a six-layer hierarchy

important aspects of this text. To describe a field of study, it is not enough to present a mass of facts and explanations. For learners to absorb, understand, and integrate this information, there must be a theme, a relationship, a thread that ties together the various parts of the narrative—in essence, a "big picture." Our big picture is Figure 1.9.

We first lay out the basic foundations of computer science (Level 1). We then proceed upward through five distinct layers of abstraction, from extremely low-level machine details such as electronic circuits and computer hardware (Level 2), through intermediate levels that address virtual machines (Level 3) and programming languages and software development (Level 4), to higher levels that investigate computer applications (Level 5), and address the use and misuse of information technology (Level 6). The material in each level provides a foundation to reveal the beauty and complexity of a higher and more abstract view of the discipline of computer science.

Laboratory Experience 1

Associated with this text is a laboratory manual that includes software packages and a collection of formal laboratory exercises. These Laboratory Experiences are designed to give you a chance to build on, modify, and experiment with the ideas discussed in the text. You are strongly encouraged to carry out these laboratories to gain a deeper understanding of the concepts presented in the chapters. Learning computer science involves not just reading and listening but also doing and trying. Our laboratory exercises will give you that chance. (In addition, we hope that you will find them fun.)

Laboratory Experience 1, titled "Building A Glossary," introduces you to this laboratory package and provides a useful tool that you may use during your study of computer science and in other courses as well. You will learn how to build a glossary of important technical terms along with their definitions and locations in the text.

Please open Laboratory Experience 1 and try it now.

EXERCISES

1. Identify some algorithms, apart from DVR instructions and cooking recipes, that you encounter in your everyday life. Write them out in any convenient notation, and explain how they meet all of the criteria for algorithms presented in this chapter.
2. A concept related, but not identical, to an algorithm is the idea of a *heuristic*. Read about heuristics and identify differences between the two. Describe a heuristic for obtaining an approximate answer to the sum of two three-digit numbers and show how this "addition heuristic" differs from the addition algorithm of Figure 1.2.
3. In the DVR instructions in Figure 1.1, Step 3 says, "Enter the channel number that you want to record and press the button labeled CHAN." Is that an unambiguous and well-defined operation? Explain why or why not.
4. Identify which type of algorithmic operation each one of the following steps belongs to:
   a. Get a value for x from the user.
   b. Test to determine if x is positive. If not, tell the user that he or she has made a mistake.
   c. Take the cube root of x.
   d. Do Steps 1.1, 1.2, and 1.3 x times.
5. Trace through the decimal addition algorithm of Figure 1.2 using the following input values:

   $$m = 3 \quad a_2 = 1 \quad a_1 = 4 \quad a_0 = 9$$
   $$b_2 = 0 \quad b_1 = 2 \quad b_0 = 9$$

   At each step, show the values for c_3, c_2, c_1, c_0, and *carry*.
6. Modify the decimal addition algorithm of Figure 1.2 so that it does not print out nonsignificant leading zeroes; that is, the answer to Exercise 5 would appear as 178 rather than 0178.
7. Modify the decimal addition algorithm of Figure 1.2 so that the two numbers being added need not have the same number of digits. That is, the algorithm should be able to add a value a containing m digits to a value b containing n digits, where m may or may not be equal to n.
8. Under what conditions would the well-known quadratic formula

   $$Roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   not be effectively computable? (Assume that you are working with real numbers.)
9. Compare the two solutions to the shampooing algorithm shown in Figures 1.3 and 1.4. Which do you think is a better general-purpose solution? Why? (*Hint:* What if you wanted to wash your hair 1,000 times?)
10. The following is Euclid's 2,300-year-old algorithm for finding the greatest common divisor of two positive integers I and J.

Step	*Operation*
1	Get two positive integers as input; call the larger value I and the smaller value J
2	Divide I by J, and call the remainder R
3	If R is *not* 0, then reset I to the value of J, reset J to the value of R, and go back to Step 2
4	Print out the answer, which is the value of J
5	Stop

   a. Go through this algorithm using the input values 20 and 32. After each step of the algorithm is completed, give the values of I, J, and R. Determine the final output of the algorithm.
   b. Does the algorithm work correctly when the two inputs are 0 and 32? Describe exactly what happens, and modify the algorithm so that it gives an appropriate error message.

11. A salesperson wants to visit 25 cities while minimizing the total number of miles she must drive. Because she has studied computer science, she decides to design an algorithm to determine the optimal order in which to visit the cities to (1) keep her driving distance to a minimum, and (2) visit each city exactly once. The algorithm that she has devised is the following:

    > The computer first lists *all* possible ways to visit the 25 cities and then, for each one, determines the total mileage associated with that particular ordering. (Assume that the computer has access to data that gives the distances between all cities.) After determining the total mileage for each possible trip, the computer searches for the ordering with the minimum mileage and prints out the list of cities on that optimal route, that is, the order in which the salesperson should visit her destinations.

    If a computer could analyze 10,000,000 separate paths per second, how long would it take to determine the optimal route for visiting these 25 cities? On the basis of your answer, do you think this is a feasible algorithm? If it is not, can you think of a way to obtain a reasonable solution to this problem?

12. One way to do multiplication is by repeated addition. For example, 47 × 25 can be evaluated as 47 + 47 + 47 + ... + 47 (25 times). Sketch out an algorithm for multiplying two positive numbers *a* and *b* using this technique.

13. A student was asked to develop an algorithm to find and output the largest of three numerical values *x, y,* and *z* that are provided as input. Here is what was produced:

    Input: *x, y, z*

    Algorithm: Check if ($x > y$) and ($x > z$). If it is, then output the value of *x* and stop. Otherwise, continue to the next line.

    Check if ($y > x$) and ($y > z$). If it is, then output the value of *y* and stop. Otherwise, continue to the next line.

    Check if ($z > x$) and ($z > y$). If it is, then output the value of *z* and stop.

    Is this a correct solution to the problem? Explain why or why not. If it is incorrect, fix the algorithm so that it is a correct solution.

14. Read about one of the early pioneers mentioned in this chapter—Pascal, Liebnitz, Jacquard, Babbage, Lovelace, Hollerith, Eckert, Mauchly, Aiken, Zuse, Atanasoff, Turing, or Von Neumann. Write a paper describing in detail that person's contribution to computing and computer science.

15. Get the technical specifications of the computer on which you are working (either from a technical manual or from your computer center staff). Determine its cost, its processing speed (in GIPS, billions of instructions per second), its computational speed (in GFlops, billions of floating point operations per second), and the size of its primary memory.

    Compare those values with what was typically available on first-, second-, and third-generation computer systems, and calculate the percentage improvement between your computer and the first commercial machines of the early 1950s.

16. A rapidly growing area of computer science is *ubiquitous computing,* in which computers automatically provide services for a user without that user's knowledge or awareness. For example, a computer located in your car contacts the garage door opener and tells it to open the garage door when the car is close to home. Read about this new model of computing and write a paper describing some of its applications. What are some of the possible problems that could result?

17. Another important new area of computer science is *cloud computing,* which relies on a computer network, along with networking software, to provide transparent access to remote data and applications. Read about

this new model of data and software access and write a paper describing some of the important uses, as well as potential risks, of this new information structure.

18. A standard computer DVD holds approximately 5 billion characters. Estimate how many linear feet of shelf space would be required to house 5 billion characters encoded as printed bound books rather than as electronic media. Assume there are 5 characters per word, 300 words per page, and 300 pages per inch of shelf.

CHALLENGE WORK

1. Assume we have a "computing agent" that knows how to do one-digit subtraction where the first digit is at least as large as the second (i.e., we do not end up with a negative number). Thus, our computing agent can do such operations as $7 - 3 = 4$, $9 - 1 = 8$, and $5 - 5 = 0$. It can also subtract a one-digit value from a two-digit value in the range 10–18 as long as the final result has only a single digit. This capability enables it to do such operations as $13 - 7 = 6$, $10 - 2 = 8$, and $18 - 9 = 9$.

   Using these primitive capabilities, design an algorithm to do *decimal subtraction* on two m-digit numbers, where $m \geq 1$. You will be given two unsigned whole numbers (a_{m-1}, $a_{m-2} \ldots a_{0)}$ and b_{m-1}, $b_{m-2} \ldots b_0$. Your algorithm must compute the value $c_{m-1}\ c_{m-2} \ldots c_0$, the difference of these two values.

$$\begin{array}{r} a_{m-1}a_{m-2}\ldots a_0 \\ -\,b_{m-1}b_{m-2}\ldots b_0 \\ \hline c_{m-1}c_{m-2}\ldots c_0 \end{array}$$

   You may assume that the top number ($a_{m-1}\ a_{m-2} \ldots a_0$) is greater than or equal to the bottom number ($b_{m-1}\ b_{m-2} \ldots b_0$) so that the result is not a negative value. However, do not assume that each individual digit a_i is greater than or equal to b_i. If the digit on the bottom is larger than the digit on the top, then you must implement a *borrowing scheme* to allow the subtraction to continue. (*Caution:* It may have been easy to learn subtraction as a first grader, but it is devilishly difficult to tell a computer how to do it!)

2. Our definition of the field of computer science is only one of many that have been proposed. Because it is so young, people working in the field are still debating how best to define exactly what they do. Review the literature of computer science (see the companion site for this text and chapter for some ideas) and browse the web to locate other definitions of computer science. Compare these definitions with the one presented in this chapter and discuss the differences among them. Discuss how different definitions may give you a vastly different perspective on the field and what people in this field do. [*Note:* A very well-known and widely used definition of computer science was presented in "Report of the ACM Task Force on the Core of Computer Science," reprinted in the journal *Communications of the ACM*, vol. 32, no. 1 (January 1989).]

3. Our focus on the history of computing looked primarily at the U.S. and the U.K.—devices like the Mark I, ENIAC, EDVAC, Colossus, ABC, and UNIVAC I. However, in a recent article by Herbert Bruderer in the *Communications of the ACM* entitled "Computing History Beyond the UK and the US: Selected Landmarks From Continental Europe" (Vol. 60, No 2, pp. 76-84), the author takes a closer look at the contributions to computer science by engineers, scientists, and mathematicians from France, Germany, Switzerland, Spain, and other European centers of learning. Take a look at that article and write a report on the fundamental contributions to computing from one specific scholar or from one specific country.