Great Ideas in Computer Science

Eric Roberts
CS 54N / CS 208E
Stanford University
Autumn 2016–17
CHAPTER 1

A Gentle Introduction

This book is about the ideas that underlie the science of computing. You won’t, however, get much out of it through reading alone. Computing, after all, is an activity. As with most activities, one learns computing best through practice. To get you started, this book provides you with the tools you need to solve simple computational problems on your own. That process of necessity involves programming, which is the process of transforming a strategy for solving a problem into a precise formulation that can be executed by a computer.

At the same time, it is hard to learn programming through the metaphorical equivalent of jumping in at the deep end of the pool. You have to approach the subject more gradually. Modern programming languages involve so many details that their complexity gets in the way of understanding the bigger picture.

To avoid overwhelming beginners with the intricacies inherent in those languages, computer science courses often introduce programming in the context of a simplified environment called a microworld. By design, microworlds are easy to learn and enable students to start programming right away. In the process, those students become familiar with the fundamental concepts of programming without having to master a lot of extraneous details.

Many different microworlds have flourished over the years, including the Project LOGO Turtle described on the next page and several more modern programming environments like Scratch and Alice. This book uses a microworld called Karel that we have used with great success in our introductory courses here at Stanford for more than 30 years. Using Karel enables you to solve challenging problems from the very beginning. And because the Karel environment encourages imagination and creativity, you can have a lot of fun along the way.
The Project LOGO Turtle

In the 1960s, Professor Seymour Papert at MIT took a language called LOGO—which he co-designed with Wally Feurzeig—and used it to teach programming to schoolchildren in the Boston area. Those students wrote LOGO programs to control a robotic “turtle” designed by Paul Wexelblat. The turtle could move forward or backward, rotate a specified number of degrees around its center, and draw pictures on large sheets of paper with a pen mounted on its underside. In 1980, Professor Papert described his experiences teaching LOGO in a wonderful book called *Mindstorms*, which offers many valuable insights into the dynamics of learning.

LOGO programs consist of a series of definitions that teach the turtle how to perform an action. The following program, for example, teaches the turtle to “square”:

```
to square
  repeat 4
    forward 40
    left 90
  end
end
```

This program draws the following picture:

Programs can then use `square` as an action, like this:

```
to flower
  repeat 36
    square
    left 10
  end
end
```

which produces the following assemblage of squares:

Cover of Seymour Papert’s *Mindstorms* and a picture from the book of children using the LOGO turtle
1.1 Introducing Karel

In the 1970s, a Stanford graduate student named Rich Pattis decided that it would be easier to teach the fundamentals of programming if students could learn those ideas in an environment free from the complexities that characterize most programming languages. Drawing inspiration from the success of the LOGO project, Pattis designed a microworld in which students teach a virtual robot to solve simple problems. Pattis called his robot Karel after the Czech playwright Karel Čapek whose 1923 play *R.U.R.* (Rossum’s Universal Robots) gave the word *robot* to the English language. Karel was an immediate success and soon spread to universities all over the world.

Programming in Karel

Karel is a very simple robot living in a very simple world. By giving Karel a set of instructions, you can direct it to perform certain tasks within its world. Those instructions constitute a program. Generically, the text that makes up a program is called code. When you write a Karel program, you must do so in a precise way so that Karel can correctly interpret it. Every program you write must obey a set of syntactic rules that define whether that program is legal.

In many respects, the rules of the Karel programming language are similar to those you will see in more sophisticated languages. The most important difference is that Karel’s programming language is tiny—so small, in fact, that it is easy to learn everything there is to know about the Karel language in less than an hour. The details are easy to master. Even so, you will discover that solving a problem in Karel’s world can be extremely challenging. Solving problems is the essence of programming. You learn the rules to unlock the problem-solving power.

Karel’s world

Karel’s world is defined by streets running horizontally (east-west) and avenues running vertically (north-south). The intersection of a street and an avenue is called a corner. Karel can only be positioned on corners and must be facing one of the four standard compass directions (north, east, south, and west). In the following sample world, Karel is facing east at the corner of 1st Street and 1st Avenue:
Several other components of Karel’s world can be seen in this example. The object in front of Karel is a beeper. According to Rich Pattis, beepers are “plastic cones which emit a quiet beeping noise.” Karel can only detect a beeper if it is on the same corner. The solid lines in the diagram are walls. Walls serve as barriers within Karel’s world. Karel cannot walk through walls and must instead go around them. Karel’s world always has walls along the edges, but the world may have different dimensions depending on the specific problem Karel needs to solve.

**Karel’s built-in functions**

The operations that Karel performs as it executes a program are called *functions*. When Karel is shipped from the factory, it knows how to execute only the following four functions:

- **move()**
  
  Karel moves forward one block. Karel cannot move forward if there is a wall blocking the way.

- **turnLeft()**
  
  Karel rotates 90 degrees to the left (counterclockwise).

- **pickBeeper()**
  
  Karel picks up one beeper from the current corner and stores that beeper in its beeper bag, which can hold an infinite number of beepers. Karel can execute the **pickBeeper** function only if there is a beeper on that corner.

- **putBeeper()**
  
  Karel puts a beeper from its bag down on the current corner. Karel can execute the **putBeeper** function only if there are beepers in its beeper bag.

The parentheses that appear in each of these examples are part of Karel’s syntax and specify that you want to perform that operation, which in programming terminology is known as *calling the function*.

Several of the built-in functions place specific restrictions on Karel’s activities. If Karel tries to do something illegal, such as moving through a wall or picking up a nonexistent beeper, an *error condition* occurs. At this point, Karel displays a message describing what went wrong and stops executing the program.

### 1.2 Teaching Karel to solve problems

For the most part, learning to program in Karel is a matter of figuring out how to use Karel’s limited set of operations to solve a specified problem. As a simple example, suppose that you want Karel to move the beeper from its initial position on 2nd Avenue and 1st Street to the center of the ledge at 5th Avenue and 2nd Street. Thus, your goal is to write a Karel program that accomplishes the task illustrated in the following before-and-after diagram:
Getting started

The first few steps in solving this problem are simple enough. You need to tell Karel to move forward, pick up the beeper, and then move forward again to reach the base of the ledge. The Karel simulator application allows you to execute instructions by typing them into an interactive window called the Karel console. Each instruction consists of the name of the function and the parentheses that specify a function call. The first three steps in the program therefore look like this:

```
> move()
> pickBeeper()
> move()
```

Executing these function calls leaves Karel in the following position:

![before](image)

From here, Karel’s next step is to turn left to begin climbing the ledge. That operation is also easy, because Karel’s set of built-in functions includes turnLeft. Calling turnLeft at the end of the preceding program leaves Karel facing north on the corner of 3rd Avenue and 1st Street. If you then call the move instruction, Karel will move north to reach the following position:

![after](image)

The next thing you need to do is get Karel to turn right so that it is again facing east. While this operation is conceptually as easy as getting Karel to turn left, there
is a slight problem: Karel’s language includes a `turnLeft` instruction, but no `turnRight` instruction. It’s as if you bought the economy model and now discover that it is missing an important feature.

At this point, you have your first opportunity to begin thinking like a programmer. You have access to a set of Karel functions, but not exactly the set you need. What can you do? Can you accomplish the effect of a `turnRight` function using only the capabilities you have? The answer, of course, is yes. You can turn right by turning left three times. After three left turns, Karel will be facing in the desired direction. The next three steps in the program might therefore be

```
turnLeft()
turnLeft()
turnLeft()
```

Although turning left three times has the desired effect, it is hardly an elegant solution. What you as the programmer want to say is

```
turnRight()
```

The only difficulty is that Karel doesn’t yet have a definition for the `turnRight` function. To use this more expressive operation in your program, you first have to teach Karel what `turnRight` means.

**Defining functions**

One of the most powerful features of the Karel programming language is the ability to define new functions. Whenever you have a sequence of Karel operations that performs some useful task—such as turning right—you can give that sequence a name. The operation of encapsulating a sequence of instructions under a new name is called **defining a function**. The format for defining a function looks like this:

```
function name() {
    statements that make up the body of the function
}
```

In this pattern, `name` represents the name you have chosen for the new function. To complete the definition, all you have to do is specify the statements between the curly braces. The only difference between the statements in a function and those you enter on the console is that each statement in a Karel function must end with a semicolon. For example, you can define the `turnRight` function as follows:
In learning about computer science, it is essential to try things out on your own. To make that process easier, this book includes puzzle boxes from time to time that give you opportunities to practice on your own. The exercise in this puzzle box, for example, is even easier than defining `turnRight`:

```javascript
function turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
}
```

Once you've defined new functions like `turnAround` and `turnRight`, you can think of them as new primitive operations, just like `move` or `turnLeft`. In a sense, defining these new functions is like buying an advanced Karel model that includes the missing operations. The programs in the rest of this chapter assume that these two functions have already been defined.

**Completing the program**

Once you have defined `turnRight`, finishing the program is straightforward. All you need to do is move forward twice, put down the beeper, and then move forward to reach the desired final state. The complete sequence of Karel operations you need to solve the program from beginning to end looks like this:

```plaintext
> move()
> pickBeeper()
> move()
> turnLeft()
> move()
> turnLeft()
> move()
> turnRight()
> move()
> move()
> putBeeper()
> move()
> move()
```

This sequence of operations forms the solution to the original problem. Instead of typing each instruction into the console, it makes sense to define a new function that contains precisely this sequence of instructions. You can then call that function
with a single name. That function definition—together with the definition of turnRight—constitutes a complete Karel program, as shown in Figure 1-1.

In addition to the definitions of the functions moveBeeperToLedge and turnRight, Figure 1-1 also includes two examples of an important programming feature called a comment, which consists of text designed to explain the operation of the program to human readers. In both Karel and the JavaScript language you will begin to learn in Chapter 2, comments begin with the characters /* and end with the characters */. The first comment describes the operation of the program as a whole; the second offers a brief description of the turnRight operation. In a program this short, such comments may seem unnecessary. As programs become more complicated, however, comments quickly become essential tools to document the program design and make it easier for other programmers to understand.

**Decomposition**

Whenever you begin the solution of a programming problem—no matter whether that program is written in Karel or a more advanced programming language—one of your first tasks is to figure out how to divide the complete problem into smaller pieces called subproblems, each of which can be implemented as a separate
function. That process is called *decomposition*. Decomposition is one of the most powerful strategies that programmers use to manage complexity, and you will see it again and again throughout this book.

To get a sense of how decomposition works in the context of a very simple problem, imagine that Karel is standing on a “road” as shown on the left side of the following before-and-after diagram:

Karel’s job is to fill each of the two potholes—the one on 2\textsuperscript{nd} Avenue and the one on 5\textsuperscript{th} Avenue—with a beeper and then continue on to the next corner, ending up in the position shown on the right.

Although you could certainly solve this problem using Karel’s four predefined instructions, you can improve the structure of your program by using functions. If nothing else, you can use the functions \texttt{turnAround} and \texttt{turnRight} to shorten the program and make its intent clearer. More importantly, you can use decomposition to break the problem down into subproblems and then solve those problems independently. You can, for example, break the problem of filling the pothole into the following subproblems:

1. Move one block forward to reach the first pothole on 2\textsuperscript{nd} Avenue.
2. Fill the pothole by dropping a beeper into it.
3. Move three blocks forward to reach the second pothole on 5\textsuperscript{th} Avenue.
4. Fill the pothole by dropping a beeper into it.
5. Move one block forward to reach the desired final position.

If you think about the problem in this way, you can use functions to that the program reflects your conception of the problem structure, as shown in Figure 1-2. The decomposition, moreover, takes advantage of the fact that the same function can be used in both steps 2 and 4 of the solution strategy.

As with any programming problem, there are other decomposition strategies you might have tried. Some strategies make the program easier to read, while others only make the meaning more opaque. As programming problems become more complex, decomposition will turn out to be one of the most important aspects of the design process.
Choosing an effective decomposition is much more of an art than a science, although you will find that you get better with practice. Section 1.4 presents some general guidelines that will help you in that process.

### 1.3 Control statements

As useful as it is, the ability to define new functions does not actually enable Karel to solve any new problems. Because each function name is merely shorthand for a specific set of instructions, it is always possible to expand a program written as a series of function calls into a single function that accomplishes the same task, although the resulting code is likely to be long and difficult to read. The instructions are still executed in a fixed order that does not depend on the state of Karel’s world. Before you can solve more interesting problems, you need to learn...
how to write programs in which this strictly linear, step-by-step order of operations does not apply. To unlock the extraordinary power that this ability provides, you need to learn several new statements in Karel’s programming language that enable Karel to examine its world and change its execution pattern accordingly.

Statements that affect the order in which a program executes instructions are called control statements. Control statements fall into the following two classes:

1. **Conditional statements.** Conditional statements specify that certain statements in a program should be executed only if a particular condition holds. In Karel, you specify conditional execution using an `if` statement.

2. **Iterative statements.** Iterative statements specify that certain statements in a program should be executed repeatedly, forming what programmers call a loop. Karel supports two iterative statements: a `repeat` statement that allows you to repeat a set of instructions a fixed number of times, and a `while` statement that allows you to repeat a set of instructions as long as some condition holds.

The sections that follow introduce each of these control statement forms in the context of a Karel problem that illustrates the need for each statement type.

### Conditional statements

To get a sense of where conditional statements might come in handy, let’s go back to the pothole-filling program presented at the end of section 1.2. Before filling the pothole in the `fillPothole` function, Karel might want to check to see if some other repair crew has already filled the hole, which means that there is already a beeper on that corner. If so, Karel does not need to put down a second one. To represent such checks in the context of a program, you need to use the `if` statement, which ordinarily appears in the following form:

```plaintext
if (conditional test) {
  statements to be executed only if the condition is true
}
```

The conditional test shown in the first line of this pattern must be replaced by one of the tests Karel can perform on its environment. The result of that conditional test is either true or false. If the test is true, Karel executes the statements enclosed in braces; if the test is false, Karel does nothing.

The tests that Karel can perform are listed in Figure 1-3. Like function calls, tests include an empty set of parentheses, which are simply part of the Karel syntax. Every test in the list is paired with a second test that checks the opposite condition. For example, you can use the `frontIsClear` condition to check whether the path ahead of Karel is clear or the `frontIsBlocked` condition to see if there is a wall
blocking the way. Choosing the right condition to use in a program requires you to think about the logic of the problem and see which condition is easiest to apply.

You can use the if statement to modify the definition of the fillPothole function so that Karel puts down a beeper only if there is not already a beeper on that corner. The new definition of fillPothole looks like this:

```plaintext
function fillPothole () {
    turnRight();
    move();
    if (noBeepersPresent()) {
        putBeeper();
    }
    turnAround();
    move();
    turnRight();
}
```

The if statement in this example illustrates several features common to all control statements in Karel. The control statement begins with a header, which indicates the type of control statement along with any additional information to control the program flow. In this case, the header is

```plaintext
if (noBeepersPresent())
```

which shows that the statements enclosed within the braces should be executed only if the noBeepersPresent test is true. The statements enclosed in braces represent the body of the control statement.
It often makes sense to include if statements in a function that check whether it makes sense to apply that function in the current state of the world. For example, calling the fillPothole function makes sense only if Karel is facing east directly above a hole. You can use the rightIsClear test to determine if there is a hole to the south, which is the direction to the right of the one that Karel is facing.

The following implementation of fillPothole includes this test along with the noBeepersPresent test you have already seen:

```javascript
function fillPothole() {
    if (rightIsClear()) {
        turnRight();
        move();
        if (noBeepersPresent()) {
            putBeeper();
        }
        turnAround();
        move();
        turnRight();
    }
}
```

As you can see from the spacing used in this example, the body of each control statement is indented with respect to the statements that enclose it. The indentation makes it much easier to see exactly which statements will be affected by the control statement. Such indentation is particularly important when the body of a control statement contains other control statements. Control statements that occur inside other control statements are said to be nested.

The outcome of a decision in a program is not always a matter of whether to do nothing or perform some set of operations. In some cases, you need to choose between two alternative courses of action. To take account of such situations, the if statement in Karel has an extended form that looks like this:

```javascript
if (conditional test) {
    statements to be executed if the condition is true
} else {
    statements to be executed if the condition is false
}
```

The decision as to whether you need a simple if statement or the if-else form depends on the nature of the problem you are trying to solve. In most cases, the English description of the problem will give you at least some clues. If the language in which you describe the problem includes the English words else or otherwise, the if-else form is more likely to capture that idea.
Iterative statements

In solving Karel problems, you will often find that repetition is a necessary part of your solution. If you were really going to program a robot to fill potholes, it would hardly be worthwhile to have it fill just one. The value of having a robot perform such a task comes from the fact that the robot could repeatedly execute its program to fill one pothole after another.

To see how repetition can be used in the context of a programming problem, consider the following stylized roadway in which the potholes are evenly spaced along 1st Street at every even-numbered avenue:

Your mission is to write a program that instructs Karel to fill all the holes in this road. Note that the road reaches a dead end after 11th Avenue, which means that you have exactly five holes to fill.

Since the problem statement tells you that there are exactly five holes to fill, it makes sense to use a repeat statement, which specifies that you want to repeat an operation a predetermined number of times. The repeat statement looks like this:

```
repeat (number of repetitions) {
    statements to be repeated
}
```

For example, if you want to change the program for filling potholes so that it solves the more complex problem of filling five evenly-spaced holes, all you have to do is define the following function:

```
function fillFivePotholes() {
    repeat (5) {
        move();
        fillPothole();
        move();
    }
}
```

The repeat statement is useful only when you know in advance the number of repetitions you need to perform. In most applications, the number of repetitions is controlled by the specific nature of the problem. For example, it seems unlikely that
a pothole-filling robot could always count on there being exactly five potholes. It would be much better if Karel could continue to fill holes until it encountered some condition that caused it to stop, such as reaching the end of the street. Such a program would be more general in its application and would work correctly in either of the following worlds as well as any other world in which the potholes were spaced exactly two corners apart:

To write a general program that works with any of these worlds, you need to use a while statement. In Karel, a while statement has the following general form:

```karel
while (conditional test) {
    statements to be repeated
}
```

The conditional test in the header is chosen from the set of conditions listed in Figure 1-3. In this case, Karel needs to check whether the path in front is clear by invoking the condition `frontIsClear`. If you use the `frontIsClear` condition in a while loop, Karel will repeatedly execute the loop until it hits a wall. The while statement therefore makes it possible to solve the somewhat more general problem of repairing a roadway, as long as the potholes appear at every even-numbered corner and the end of the roadway is marked by a wall. The following definition of the function `fillRegularPotholes` accomplishes this task:

```karel
function fillRegularPotholes() {
    while (frontIsClear()) {
        move();
        fillPothole();
        move();
    }
}
```

Solving general problems

So far, the various pothole-filling programs have not been very realistic, because they rely on specific conditions—such as evenly spaced potholes—that are unlikely to be true in the real world. If you want to write a more general program to fill potholes, it should be able to work with fewer constraints. In particular,
• *The program should be able to work with roads of arbitrary length.* It does not make sense to design a program that works only for roads with a predetermined number of corners. Instead, you want to make the same program work for roads of any length.

• *The potholes may occur at any position in the roadway.* There should be no limits on the number of potholes or any restrictions on their spacing. A pothole is identified simply by an opening in the wall representing the road surface.

To change the program so that it solves this more general problem requires you to think about the overall strategy in a different way. Instead of having a loop that cycles through each pothole, you need to have it call `fillPothole` at every intersection along the roadway.

This strategic analysis suggests that the solution to the general problem might be as simple as the following definition:

```
function fillAllPotholes() {
    while (frontIsClear()) {
        fillPothole();
        move();
    }
}
```

Unfortunately, the solution is not quite so easy. The program as written contains a logical flaw—the sort of error that programmers call a *bug.* This book uses the bug symbol on the right to mark functions that contain errors to ensure that you don’t accidentally use those examples as models for your own code.

The bug in this example turns out to be relatively subtle. It would be easy to miss, even if you thought you had tested the program thoroughly. In particular, the program works correctly on all the pothole-filling worlds you’ve seen so far and on many which you haven’t. It only fails if there is a pothole in the very last avenue on the street, as illustrated by the following before-and-after diagram:

```
before

after
```

In this example, Karel stops without filling the last pothole. In fact, if you watch the execution carefully, Karel never even goes down into that last pothole to check whether it needs filling. What’s the problem here?
If you follow through the logic of the program carefully, you’ll discover that the
bug lies in the structure of the loop in `fillAllPotholes`, which looks like this:

```java
while (frontIsClear()) {
    fillPothole();
    move();
}
```

As soon as Karel finishes filling the pothole on 6th Avenue, it executes the `move`
instruction and returns to the top of the `while` loop. At that point, Karel is standing
at the corner of 7th Avenue and 2nd street, where it is up against the boundary wall.
Because the `frontIsClear` test now fails, the `while` loop exits without checking
the last segment of the roadway.

The bug in this program is an example of a programming problem called a
**fencepost error.** The name comes from the fact that it takes one more fence post
that you might think to fence off a particular distance. How many fence posts, for
example, do you need to build a 100-foot fence if the posts are always positioned 10
feet apart? The answer is 11, as illustrated by the following diagram:

```
+---------------------+
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
+---------------------+ 100 feet, 11 fenceposts

The situation in Karel’s world has much the same structure. In order to fill potholes
in a street that is seven corners long, Karel has to check for seven potholes but only
has to move six times. Because Karel starts and finishes at an end of the roadway, it
needs to execute one fewer `move` instruction than the number of corners it checks.

Once you discover it, fixing this bug is actually quite easy. Before Karel stops at
the end of the roadway, all that the program has to do is to make a special-case
check for a pothole at the final intersection, as follows:

```java
function fillAllPotholes() {
    while (frontIsClear()) {
        fillPothole();
        move();
    }
    fillPothole();
}
```

The complete program appears in Figure 1-4.
When you are faced with a complex programming problem, figuring out how to decompose the problem into pieces is usually one of your most important tasks. One of the most productive strategies is called **stepwise refinement**, which consists of solving problems by starting with the problem as a whole. You break the whole problem down into pieces, and then solve each piece, breaking those down further if necessary.

### An exercise in stepwise refinement

Suppose that Karel is initially facing east at the corner of 1st Street and 1st Avenue in a world in which each avenue may contain a vertical tower of beepers of an unknown height, although some avenues may also be empty. Karel’s job is to
collect the beepers in each of these towers, put them all back down on the easternmost corner of 1st Street, and then return to its starting position. Figure 1-5 illustrates the operation of this program for one possible world.

The key to solving this problem is to decompose the program in the right way. This task is more complex than the others you have seen, which makes choosing appropriate subproblems more important to obtaining a successful solution.

**The principle of top-down design**

The central idea in stepwise refinement is that you should start the design of your program from the top, which refers to the level of the program that is conceptually highest and most abstract. At this level, the beeper tower problem is clearly divided into three independent phases. First, Karel has to collect all the beepers. Second, Karel has to deposit them on the last intersection. Third, Karel has to return to its home position. This outline suggests the following decomposition of the problem:

```plaintext
function collectBeeperTowers() {
    collectAllBeepers();
    dropAllBeepers();
    returnHome();
}
```

At this level, the problem is easy to understand. Even though you have not written the code for the functions in the body of `collectBeeperTowers`, it is important to convince yourself that, as long as you believe that the functions you are about to write will solve the subproblems correctly, you will have a solution to the problem as a whole.

![Figure 1-5: Before-and-after diagrams for the CollectBeeperTowers problem](image)
Refining the first subproblem

Now that you have defined the structure for the program as a whole, it is time to move on to the first subproblem, which consists of collecting all the beepers. This task is itself more complicated than the problems you have seen so far. Collecting all the beepers means that you have to pick up the beepers in every tower until you get to the final corner. The fact that you need to repeat an operation for each tower suggests that you need to use a while loop.

But what does this while loop look like? First of all, you should think about the conditional test. You want Karel to stop when it hits the wall at the end of the row. Thus, you want Karel to keep going as long as the space in front is clear. Thus, you know that the collectAllBeepers function will include a while loop that uses the frontIsClear test. At each position, you want Karel to collect all the beepers in the tower beginning on that corner. If you give that operation a name like collectOneTower, you can then write a definition for the collectAllBeepers function even though you haven’t yet filled in the details. You do, however, have to be careful. To avoid the fencepost problem described on page 17, the code must call collectOneTower after the last cycle of the loop, as follows:

```plaintext
function collectAllBeepers {
   while (frontIsClear()) {
      collectOneTower();
      move();
   }
   collectOneTower();
}
```

As you can see, this function has the same structure as the fillAllPotholes function in Figure 1-4. The only difference is that collectAllBeepers calls collectOneTower where the earlier one called fillPothole. These two programs are each examples of a general strategy that looks like this:

```plaintext
while (frontIsClear()) {
   Perform some operation.
   move();
}
Perform the same operation for the final corner.
```

You can use this strategy whenever you need to perform an operation on every corner as you move along a path that ends at a wall. If you remember the general strategy, you can quickly write the code whenever you encounter a problem of a similar form. Reusable strategies of this sort come up frequently in programming and are referred to as programming idioms or patterns. The more patterns you know, the easier it will be for you to find one that fits a particular type of problem.
Coding the next level

Even though the code for `collectAllBeepers` is complete, you can’t run the program until you implement `collectOneTower`. When `collectOneTower` is called, Karel is standing either at the base of a tower or on an empty corner. In the former case, you need to collect the beepers in the tower. In the latter case, you can simply move on. This situation at first suggests that you need an `if` statement in which you call `beepersPresent` to see whether a tower exists.

Before you add such a statement to the code, it is worth giving some thought to whether you need to make this test. Often, programs can be made much simpler by observing that cases that at first seem to be special can be treated in precisely the same way as the more general situation. In the current problem, what happens if you decide that there is a tower of beepers on every avenue but that some of those towers are zero beepers high? Making use of this insight simplifies the program because you no longer have to test whether there is a tower on a particular avenue.

The `collectOneTower` function is still complex enough that an additional level of decomposition makes sense. To collect all the beepers in a tower, Karel has to climb the tower to collect each beeper, turn around, and then return to the wall that marks the southern boundary of the world. These steps suggest the following code:

```javascript
function collectOneTower() {
    turnLeft();
    collectLineOfBeepers();
    turnAround();
    moveToWall();
    turnLeft();
}
```

The `turnLeft` instructions at the beginning and end of the `collectOneTower` function are critical to the correctness of this program. When `collectOneTower` is called, Karel is always somewhere on 1st Street facing east. When it completes its operation, the program as a whole will work correctly only if Karel is again facing east at that same corner.

Finishing up

Although the hard work has been done, there are still several loose ends that need to be resolved. The main program calls two functions—`dropAllBeepers` and `returnHome`—that are as yet unwritten. Similarly, `collectOneTower` calls `collectLineOfBeepers` and `moveToWall`. Fortunately, each of these functions is simple enough to code without any further decomposition. A complete implementation of the `CollectBeeperTowers` program appears in Figure 1-6 on the next two pages.
/*
 * File: CollectBeeperTowers.k
 * -----------------------
 * This program collects all the beepers in a series of towers, deposits
 * them at the easternmost corner on 1st Street, and then returns home.
 */

function collectBeeperTowers() {
    collectAllBeepers();
    dropAllBeepers();
    returnHome();
}

/* Collects the beepers from every tower along 1st Street */

function collectAllBeepers() {
    while (frontIsClear()) {
        collectOneTower();
        move();
    }
    collectOneTower();
}

/* Collects the beepers in a single tower */

function collectOneTower() {
    turnLeft();
    collectLineOfBeepers();
    turnAround();
    moveToWall();
    turnLeft();
}

/* Collects a consecutive line of beepers */

function collectLineOfBeepers() {
    while (beepersPresent()) {
        pickBeeper();
        if (frontIsClear()) {
            move();
        }
    }
}

/* Drops all the beepers from Karel's bag on the current corner */

function dropAllBeepers() {
    while (beepersInBag()) {
        putBeeper();
    }
}
Although top-down design is a critical strategy for programming, it cannot be applied mechanically without thinking about problem-solving strategies. Figuring out how to solve a particular problem generally requires considerable creativity.

Suppose, for example, that you want to teach Karel to escape from a maze. In Karel’s world, a maze might look like this:

Karel’s job is to navigate the corridors of the maze until it finds the beeper indicating the exit. The program, however, must be general enough to solve any maze, and not just the one pictured here.

For most mazes, you can use a simple strategy called the **right-hand rule**, in which you start by putting your right hand on the wall and then go through the maze without ever taking your hand off the wall. Another way to express this strategy is

```javascript
/* Returns Karel to the corner of 1st Avenue and 1st Street, facing east */
function returnHome() {
    turnAround();
    moveToWall();
    turnAround();
}

/* Moves Karel forward until it is blocked by a wall */
function moveToWall() {
    while (frontIsClear()) {
        move();
    }
}
```
to proceed through the maze one step at a time, always taking the rightmost available path.

The program that implements the right-hand rule turns out to be easy to implement in Karel and fits in a single function:

```javascript
function solveMazeUsingRightHandRule() {
    while (noBeepersPresent()) {
        turnRight();
        while (frontIsBlocked()) {
            turnLeft();
        }
        move();
    }
}
```

At the beginning of the outer while loop, Karel turns right to check whether that path is available. The inner while loop then turns left until an opening appears. When that happens, Karel moves forward, and the entire process continues until Karel reaches the beeper marking the end of the maze.

Write a Karel program that creates a checkerboard pattern of beepers inside an empty rectangular world. For example, your program should produce the pattern on the right if you run it in an empty 8×8 world. Your program may leave Karel on any convenient square when the program is finished.

As you think about how to solve the problem, you should make sure to use a strategy that works with checkerboards that have a different size from the 8×8 world shown in the example. In particular, you should make sure that your program generates a checkerboard pattern in worlds with an odd number of streets or avenues.
In many ways, computer science is an unfortunate name for the discipline that this book seeks to describe. The word science traditionally refers to the study of natural phenomena. When people talk about biological science or physical science, that usage feels right. Computers, however, are the products of human technology rather than something that exists in the natural world, which makes the word science seem less appropriate. Human technology has also produced cars, but no one uses the term car science. People refer instead to automotive engineering or automobile technology. Why should computers be any different?

To answer this question, it is important to recognize that the computer itself is only part of the story. Your laptop, tablet, and smart phone are all examples of computer hardware. They are tangible. You can buy one, take it home, and put it on your desk or in your pocket. If it is large enough, you could use it as a doorstop, albeit a rather expensive one. But if there were nothing there besides the hardware, being a doorstop would be one of the few jobs it could do. Getting a computer to do anything useful also requires software, which is the generic term for the programs that control the hardware and enable it to perform computational tasks.

In contrast to hardware, software is an abstract, intangible entity. As you will discover in Chapter 6, software is represented internally as a sequence of simple instructions in a language that the hardware can interpret. Computer science today is concerned primarily with software and, more importantly, with the even more abstract domain of problem solving. Problem solving turns out to be a highly challenging activity that requires creativity, skill, and discipline. For the most part, computer science is best thought of as the science of problem solving in which the solutions happen to involve a computer.
The mathematical contributions of Muhammed ibn Mūsā al-Khwārizmī

After the fall of the Roman Empire in the 4th century BCE, European mathematics entered a dark age that lasted more than 750 years. During that time, the discoveries of classical mathematics were kept alive primarily in the Persian Empire. At the beginning of the 9th century, Caliph Harun al-Rashid and his son al-Ma'mun established the House of Wisdom in Baghdad, which soon became the intellectual center of the Islamic world.

One of the important mathematicians at the House of Wisdom was Muhammed ibn Mūsā al-Khwārizmī, who wrote the first systematic treatment of algebra. His name survives in two English words: the word algebra comes from the title of most famous work, al-Kitāb al-mukhtaṣar fī hishāb al-jabr wal-muqābala, and the word algorithm derives from al-Khwārizmī’s name.

In the 12th century, the English scholar Robert of Chester translated al-Khwārizmī’s treatise from Arabic into Latin, giving it the title Liber Algebræ et Almucabola. The publication of Robert’s translation brought the ideas of algebra and geometry back into European thought, where they had enormous influence in the centuries that followed.

Statue of the Persian mathematician Muhammed ibn Mūsā al-Khwārizmī outside the gates of Khiva, Uzbekistan
If computer science is the discipline of solving problems with the aid of a computer, anyone seeking to study computer science must first understand an idea that is fundamental to the discipline of problem solving, which is the concept of an algorithm. Informally, an algorithm is simply a strategy for solving a problem. To appreciate how computer scientists use the term, however, it is useful to formalize that intuitive understanding and tighten up the definition.

To be considered an algorithm, a solution strategy must fulfill three basic requirements. First of all, an algorithm must be presented in an unambiguous form that makes it clear to the reader precisely what steps are involved. Second, the steps within an algorithm must be effective, in the sense that it is possible to carry them out in practice. Third, an algorithm must not run on forever but must deliver its answer in a finite amount of time. In summary, an algorithm must be

1. Clearly and unambiguously defined.
2. Effective, in the sense that its steps are executable.
3. Finite, in the sense that it terminates after a bounded number of steps.

### 2.1 The Babylonian square-root algorithm

As described on the preceding page, the word algorithm comes from the name of the Persian mathematician Muḥammad ibn Mūsā al-Khwārizmī. Al-Khwārizmī’s work dates to the 9th century, but the first algorithms worthy of the name are much older than that. Almost 4000 years ago, Babylonian mathematicians clearly used an algorithmic process to calculate square roots, even though the details of that process are recorded only in later sources.

The evidence that early Babylonian mathematicians used an algorithmic process to calculate square roots comes from cuneiform tablets such as the one shown in Figure 2-1, which shows an approximation of the square root of 2 that is far more accurate than anyone could possibly derive through measurement alone. And although the details of the Babylonian method for computing square roots have been lost over time, historians of mathematics believe that their technique was similar to the algorithm described by the certainly 1st-century Greek mathematician Hero of Alexandria. The algorithm Hero described is usually called the Babylonian method after its most likely origin.

The Babylonian method for calculating square roots is an example of a general technique called successive approximation, in which you begin by making a rough guess at the answer and then improve that guess through a series of refinements that get closer and closer to the exact answer. For example, if you want to find the square root of some number \( n \), you start by choosing some smaller number \( g \) as your first guess. At every point in the process, your guess \( g \) will be smaller or larger than
the actual square root. In either case, if you divide \( n \) by \( g \), the result will inevitably lie on the opposite side of the desired value. For example, if \( g \) is too small, \( n \) divided by \( g \) will be too large, and vice versa. Averaging the two values will always give a better approximation. At each step, you simply replace your previous guess \( g \) by the result of the following formula, which averages \( g \) and \( n \) divided by \( g \):

\[
\frac{g + \frac{n}{g}}{2}
\]

You then continue to apply this formula to each new guess until the answer is as close to the actual value as you need it to be.

To get more of a sense of how the Babylonian method works, it helps to consider a simple example. Suppose that you want to calculate, as the scribes who incised the cuneiform tablet did, the square root of 2. One possible first guess for \( g \) is 1, which is half the value of \( n \). The first approximation step therefore computes the following average:

\[
\frac{1 + \frac{2}{1}}{2} = \frac{3}{2} = 1.5
\]
The value 1.5 is closer to the actual square root of 2—which is approximately 1.4142136—so the process is on the right track.

To calculate the next approximation, all you need to do is plug $\frac{3}{2}$ into the formula as the next value of $g$, and calculate the new average, as follows:

$$\frac{\frac{1}{3} + \frac{2}{3}}{2} = \frac{\frac{17}{12}}{2} \approx 1.416667$$

From here, you simply repeat the calculation with $\frac{17}{12}$ as the new value of $g$:

$$\frac{\frac{17}{12} + \frac{2}{\frac{17}{12}}}{2} = \frac{\frac{577}{408}}{2} \approx 1.4142157$$

Applying successive approximation one more time gives

$$\frac{\frac{577}{408} + \frac{2}{\frac{577}{408}}}{2} = \frac{\frac{665857}{470832}}{2} \approx 1.4142136$$

After just four cycles, the Babylonian method has produced an approximation to the square root of 2 that is correct to eight decimal digits. Moreover, because each step generates an approximation that is closer to the exact value, you can repeat the process to produce an approximation at any desired level of accuracy.

Even though the Babylonians managed to work through this calculation with far less mathematical training that one gets in school today, readers who are out of practice are likely to find that their eyes glaze over when they encounter all these fractions. It is easy to accept that the final result is a good approximation of the square root of 2, but it is equally easy to skip over all the calculations that led to that answer. You will, however, get more out of this example if you try it yourself.

| Use the Babylonian method to calculate the square root of 3 so that your answer has at least six digits of accuracy. How many steps does it take to reach that point? |

Unfortunately, the Babylonian method is not yet an algorithm because it fails to satisfy the condition that an algorithm must terminate eventually. No matter how many steps you take, using the Babylonian method to compute the square root of 2 can never generate an exact answer because no exact answer exists. In the 3rd century BCE, the Greek mathematician Euclid proved that the square root of 2 is irrational, which means that it cannot be expressed as the quotient of two integers.
or as a finite decimal expansion. The Babylonian method therefore becomes an algorithm only if you specify some way to stop the process when the answer becomes sufficiently close. Determining when to stop depends on how accurate you need the answer to be. For the Babylonians, three successive approximation steps produced a sufficiently accurate answer; modern computers make it easy to generate even closer approximations quickly. Turning the Babylonian method into an algorithm requires providing a precise definition of when the current approximation is close enough to allow the process to stop.

## 2.2 Programming simple calculations

If you worked through the puzzle box on the previous page, you presumably came to realize that arithmetic calculations—even in a computation this short—quickly become tedious. Computers are much better than people at arithmetic calculations, and it would be wonderful if you could harness that computational power to help you with this sort of problem. To do so, however, you will first need to learn a little more about programming.

The programming examples in this book use a language called **JavaScript**, which is today the most popular language for writing interactive web applications. JavaScript was developed in 1995—reportedly in just ten days—by Brendan Eich at the Netscape Communications Corporation. Because of its popularity, JavaScript is implemented in every major web browser, which means that any device with a web browser can run JavaScript programs without any additional software. The focus of this book, however, is not on the JavaScript language itself but rather on the algorithms that you write using that language. This book presents only a subset of JavaScript that is sufficient to solve the programming problems that appear in the puzzle boxes. Moreover, the book introduces new features of JavaScript only when you need them in the context of a particular problem.

When you launch the SJS web application, one of the windows that appears is the **JavaScript console**, which gives you an interactive window where you can enter calculations and see the result. For example, if you want to show that two plus two has the value four (which is the first calculation that Bill Gates and Paul Allen entered into their implementation of Basic that launched what would become Microsoft), all you need to do is type `2 + 2` into the JavaScript console, as follows:

<table>
<thead>
<tr>
<th>JavaScript Console</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt; 2 + 2</code></td>
</tr>
<tr>
<td><code>4</code></td>
</tr>
</tbody>
</table>

Similarly, if you want to subtract 173 from 342 (which happens to be the problem the singer-songwriter, satirist, and mathematician Tom Lehrer proposed in his song
“New Math” in 1965), you can simply type that the following expression on the JavaScript console:

```
> 342 - 173
169
```

### Arithmetic expressions

The calculations specified by the inputs $2 + 2$ and $342 - 173$ are both examples of arithmetic expressions, which are sequences of individual values called terms combined using symbols called operators. JavaScript’s operators—most of which are familiar from elementary-school arithmetic—include the following:

- **Addition** (+)
- **Subtraction** (or negation, if written with no value to its left) (−)
- **Multiplication** (*)
- **Division** (/)
- **Remainder** (%)

Following standard mathematical conventions, the multiplication, division, and remainder operations are performed before addition and subtraction, although you can use parentheses to change the evaluation order. For example, if you want to average the numbers 4 and 7, you simply enter the following expression in the JavaScript console:

```
> (4 + 7) / 2
5.5
```

It is important to note that the parentheses are important here. If you leave them out, JavaScript divides 7 by 2 and then adds the result to 4. Performing the operations in that order produces 7.5, which is not the correct average.

### Variables

When you write a program, it is often convenient to use symbolic names to refer to a value that can change as the program runs. In programming, symbolic names that refer to such values are called variables. Every variable in JavaScript has two attributes: a **name** and a **value**. To understand the relationship of these attributes, it is best to think of a variable as a box with a label attached to the outside. The name of the variable appears on the label and is used to tell different variables apart. The value corresponds to the contents of the box. The name of the box is fixed, but you can change the value as often as you like.
To create a new variable in JavaScript, the usual approach is to include a line in your program that begins with the keyword `var` followed by the name of the variable, an equal sign, the initial value for that variable, and finally a semicolon. A program line that introduces a new variable is called a declaration. The following declaration, for example, introduces a variable named `r` and assigns it the value 10:

```javascript
var r = 10;
```

Conceptually, this declaration creates a box inside the computer’s memory, gives it the label `r`, and stores the value 10 in the box, like this:

```
r
10
```

You can make your programs more readable by using variable names that immediately suggest the meaning of that variable. If `r`, for example, refers to the radius of a circle, that name is perfectly appropriate. In most cases, however, it is better to use longer names that make it clear to anyone reading your program exactly what value that variable contains. For example, if you needed a variable to keep track of the number of pages in a document, it would be better to use a name like `numberOfPages` than to use a shorter, more cryptic name like `np`.

The variable name `numberOfPages` may at first look a little odd because of the capital letters that appear in the middle of the name. That name, however, follows what has become a widely accepted standard for naming variables. By convention, variable names in JavaScript begin with a lowercase letter but include uppercase letters at the beginning of each new word. This convention is called *camel case* because it creates uppercase “humps” in the middle of the variable name.

**Assignment**

Once you have declared a variable, you can change its value using an assignment statement, which looks just like a declaration without the `var` keyword at the beginning. For example, if you execute the assignment statement

```javascript
r = 2.5;
```

the value in the variable `r` would change as follows:

```
r
2.5
```

Assignment statements are often used to modify the current value of a variable. For example, you could add the value of `deposit` to `balance` using the statement

```javascript
balance = balance + deposit;
```
which takes the current value of balance, adds the value of deposit, and then stores the result back in balance. Assignment statements of this form are so common that JavaScript allows you to use the following shorthand

```javascript
balance += deposit;
```

Similarly, you can subtract the value of surcharge from balance by writing

```javascript
balance -= surcharge;
```

More generally, the JavaScript statement

```javascript
variable op = expression;
```

is equivalent to

```javascript
variable = variable op (expression);
```

where the parentheses are included to emphasize that the entire expression is evaluated before op is applied. Such statements are called shorthand assignments.

**Constants**

In many applications, it is useful to give names to values that remain unchanged throughout the program. Such values are called constants. As an example, if you are performing geometrical calculations involving circles, it is useful to have a constant named PI whose value is an approximation of the mathematical constant $\pi$. Although you will soon discover that PI is already defined in one of the standard libraries, you could always define it yourself by writing the following declaration:

```javascript
var PI = 3.14159265;
```

By convention, constant names are written entirely in uppercase using underscores to indicate word boundaries.

**Sequential calculations**

The ability to define variables and constants makes it possible to express more meaningful calculations, even in the JavaScript console. The following console session, for example, calculates the area of a circle of radius 10:
JavaScript does not include an operator for raising a number to a power, so the computation of $r^2$ simply multiplies $r$ by itself.

### 2.3 Writing simple functions

As you discovered when you wrote simple Karel programs in Chapter 1, you don’t need to type all the steps in the console window but can instead store those steps as a function. JavaScript has functions too, and they serve much the same purposes in the two languages. For example, once you have defined a function, you can call it from different parts of your program without having to copy any code. You can also use stepwise refinement to decompose complex problems into progressively simpler ones.

The most important difference between functions in Karel and JavaScript is that JavaScript functions can use information supplied by their callers and then give back information in return. The caller sends information to the function by specifying values inside the parentheses that indicate a function call. These values are called arguments. Inside the function, each of these arguments is assigned to a variable called a parameter. The function uses these parameters to compute a result, which is delivered back to the caller. This process is called returning a result and is implemented in JavaScript using a return statement.

In the context of a programming language like JavaScript, the term function is intended to evoke the similar concept in mathematics. A mathematical function like

$$f(x) = x^2 - 5$$

expresses a relationship between the value of $x$ and the value of the function. This relationship is depicted in the graph to the left, which shows how the value of the function changes with respect to the value of $x$.

In JavaScript, you can implement this function as follows:

```javascript
function f(x) {
    return x * x - 5;
}
```

In this definition, `x` is the parameter variable, which is set by the argument passed by the caller. For example, if you were to call `f(2)`, the variable `x` would be set to the value 2. The `return` statement specifies the computation needed to calculate the result. Multiplying `x` by itself gives the value 4; subtracting 5 gives the final result of −1, which is passed back to the caller.

Once you have defined this function, you can call it by executing commands in the JavaScript console, as follows:
In a similar way, you can store the steps that appear in the console window at the end of the preceding section to create a function that calculates the area of a circle given its radius:

```javascript
var PI = 3.14159265;
function circleArea(r) {
    return PI * r * r;
}
```

To call the `circleArea` function, all you need to do is specify a value for the radius. For example, given these definitions of `PI` and `circleArea`, you can then execute the following command in the JavaScript console:

Functions in JavaScript can take any number of arguments, separated from one another by commas. For example, the following function calculates the average of two arguments, `x` and `y`:

```javascript
function average(x, y) {
    return (x + y) / 2;
}
```

Parameter variables are initialized according to the order in which they appear in the call. Thus, if you call `average(4, 7)`, the parameter variable `x` will be set to 4, and the parameter variable `y` will be set to 7.

Parameter variables and any variables declared inside the body of a function are accessible only from inside that function. For this reason, those variables are called `local variables`. By contrast, variables declared outside of any function or in the console window are `global variables`, which can be used anywhere in the program.
As programs get larger and more sophisticated, using global variables makes those programs more difficult to read and maintain. The programs in this book therefore avoid using any global variables except for constants. Thus, a global definition of a constant like \( \pi \) is acceptable, but any variable whose value might change will always be declared inside a function.

(a) Write a function `convertInchesToCentimeters` that takes a distance expressed in inches and returns the corresponding value in centimeters by using the fact that one inch is equivalent to 2.54 centimeters.

(b) Write two functions, `convertFahrenheitToCelsius` and `convertCelsiusFahrenheit`, that convert temperatures between the two scales according to the following formulas:

\[
C = \frac{5}{9} (F - 32) \\
F = \frac{9}{5} C + 32
\]

Your program should allow you to produce the following console output:

```
> convertFahrenheitToCelsius(32)
0
> convertCelsiusToFahrenheit(100)
212
> convertFahrenheitToCelsius(98.6)
37
> convertCelsiusToFahrenheit(20)
68
> convertFahrenheitToCelsius(-40)
-40
> 
```

### 2.4 Using control statements in functions

Just like functions in Karel, functions in JavaScript can include control statements that affect the order of operation. The `if` and the `while` statements are essentially the same in both languages, but JavaScript uses a more flexible statement called `for` to achieve the effect of the `repeat` statement in Karel.

The major difference between control statements in the two languages lies in the conditions you can check. Karel’s conditional expressions, which all have names like `frontIsClear` and `facingNorth`, make sense only in Karel’s world. JavaScript offers a much richer set of conditions and even allows you to define your own.
Conditional expressions in JavaScript

In JavaScript, the simplest conditional expressions are those that compare two data values. You might want, for example, to determine whether two values are equal or if one is greater than or less than another. Traditional mathematics uses the operators $=, \neq, <, \leq, >, \text{and } \geq$ to signify the relationships equal to, not equal to, less than, greater than, less than or equal, and greater than or equal, respectively. Because some of these symbols don’t appear on a standard keyboard, programming languages typically write these operators in a slightly different form. In particular, JavaScript uses the operators $===, !==, <, \leq, >, \text{and } \geq$ to signify these relationships. Collectively, these operators are called relational operators because they test the relationship between two values. Like the arithmetic operators, relational operators appear between the two values to which they apply. For example, if you need to check whether the value of the $x$ is less than 0, you can use the expression $x < 0$.

At first glance, the relational operators $==$ and $!==$ are likely to appear a bit strange. Because the single equal sign had already been reserved to indicate assignment, the designers of the C programming language from which JavaScript is derived introduced a new operator consisting of two adjacent equal signs to specify equality. The inventors of JavaScript retained the $==$ operator, but defined it in such a confusing way that it is hard for anyone—novices and experienced programmers alike—to use it correctly. Many JavaScript experts argue against using the $==$ operator at all because what you almost always want in practice is to check whether the values on each side of the operator are the same. In JavaScript, the operator that checks for exact equality is $===$, and you will avoid many problems if you get into the habit of using $===$ and $!==$ in preference to the shorter forms.

The relational operators produce a result, which is either true or false. These two values are examples—and are in fact the only two examples—of Boolean data, named after the 19th-century English mathematician George Boole who developed an algebraic system for working with logic. Boolean data is extremely important in computer science, and you will be much more successful as a programmer if you come to think of true and false as values in exactly the same way you think of 17 or 42 as values.

In addition to the relational operators, JavaScript defines three logical operators that take Boolean values and combine them to form other Boolean values:

1. Logical not (true if the expression that follows is false)
2. Logical and (true if both expressions are true)
3. Logical or (true if either or both expressions are true)
The if statement

The `if` statement in JavaScript appears in the same two forms as it does in Karel. If you want to execute some code only if a particular condition applies, you use the simple `if-then` form.

```javascript
if (condition) {
    statements to be executed if the condition is true
}
```

If the problem you are trying to solve instead involves a choice between two independent sets of actions, you can use the `if-then-else` form, as follows:

```javascript
if (condition) {
    statements to be executed if the condition is true
} else {
    statements to be executed if the condition is true
}
```

For example, you can use the `if-then-else` form to implement a function called `abs`, which returns the absolute value of its argument, like this:

```javascript
function abs(x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```

Similarly, the following function returns the smaller of the two values `x` and `y`:

```javascript
function min(x, y) {
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

(a) Using the `min` function as a model, implement a function `max(x, y)` that returns the larger of `x` and `y`.

(b) Write a function `max3(x, y, z)` that returns the largest of the three arguments `x`, `y`, and `z`. 
The while statement

The while statement in JavaScript also has the same form as its counterpart in Karel and therefore looks like this:

```javascript
while (condition) {
    statements
}
```

When a JavaScript program encounters a `while` statement, it evaluates `condition` to see whether it is `true` or `false`. If it is `false`, the loop terminates and the program continues with the next statement after the entire loop. If the condition is `true`, JavaScript executes the statements in the loop body and then goes back to the beginning of the loop to check the condition again. A single pass through the statements in the body constitutes a cycle of the loop.

The `while` statement makes it possible to implement a function called `sqrt` that uses the Babylonian method described in section 2.1. The general outline of the `sqrt` function looks like this:

```javascript
function sqrt(n) {
    var g = some initial guess for the square root;
    while (the guess is not yet close enough to the desired value) {
        Set g to the average of g and n / g;
    }
    return g;
}
```

This outline consists of a JavaScript function in which several parts of the definition are replaced by English descriptions of what belongs in those places. Partial implementations of this form are called pseudocode.

Completing the missing details of the pseudocode version is not too difficult. The following statement, for example, chooses an initial guess for `g` by dividing `n` by 2, which works as well as any other starting point:

```javascript
var g = n / 2;
```

Similarly, updating the guess by averaging `g` and `n / g` is simply a matter of writing out the assignment statement so that it computes the average, which is much easier to read if you call the `average` function defined on page 35:

```javascript
g = average(g, n / g);
```

The only step in the pseudocode that requires a little thought is the test in the `while` statement, which calls for repeating the process as long as the guess is not
yet close enough to the desired value. As the programmer, you have to decide what “close enough” means. One approach is to define a constant named TOLERANCE and then check whether the square of the guess \( g \) does not deviate more than TOLERANCE from the value of \( n \).

Filling in these details makes it possible to write the complete version of the square root program, which appears in Figure 2-2. That program makes it easy to

![JavaScript program to compute square roots using the Babylonian algorithm](image-url)

```javascript
/*
 * File: BabylonianSquareRoot.sjs
 * -----------------------------
 * This program calculates square roots using the Babylonian algorithm.
 */

/* Define a constant specifying how close the value needs to be */
var TOLERANCE = 0.00000000000001;

/*
 * Calculates the square root of \( n \) using the Babylonian algorithm, which
 * operates as follows:
 * 1. Choose a guess \( g \) (any value will do; this code uses \( n / 2 \)).
 * 2. Compute a new guess by averaging \( g \) and \( n / g \).
 * 3. Repeat step 2 until the error is less than the desired tolerance.
 */

function sqrt(n) {
    var g = n / 2;
    while (abs(n - g * g) > TOLERANCE) {
        g = average(g, n / g);
    }
    return g;
}

/* Returns the absolute value of \( x \) */
function abs(x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}

/* Returns the average of \( x \) and \( y \) */
function average(x, y) {
    return (x + y) / 2;
}
calculate accurate approximations of the square root function, as shown in the following console script:

```
> sqrt(2)
1.414213562373095
> sqrt(3)
1.7320508075688772
> sqrt(49)
7
```

Write a new version of the `sqrt` function that doesn’t use a constant like `TOLERANCE` but instead continues as long as the error is getting smaller. Implementing this strategy will require you to keep track of the error from the previous cycle and then use the `while` condition to compare the current and previous value. Eventually, the process will reach the limit of machine accuracy and stop.

**The for statement**

The Karel language includes a `repeat` statement that makes it possible to execute a sequence of statements a predetermined number of times. JavaScript uses the `for` statement to accomplish the same goal, but does so in a way that is significantly more flexible and powerful.

Although you will see many other uses of `for` as you make your way through this book, most applications of the `for` statement follow one of two idiomatic patterns. The first is used when you want to perform an operation a predetermined number of times, represented by `n` in the following pattern:

```
for (var index = 0; index < n; index++)
```

The variable indicated by `index` in this pattern is called an **index variable**. Although you can use any legal variable name for this purpose, both programmers and mathematicians have a long tradition of using single-letter variable names taken from the middle of the alphabet, such as `i`, `j`, `k`, and so on. Whenever you see the variable `i` or `j` in a `for` loop, you can be reasonably confident that the variable is counting through some sequence of values.

The second idiomatic pattern appears when you want to count from one value to another. This pattern has the following general form:

```
for (var index = start; index <= finish; index++)
```
In this pattern, the body of the for loop is executed with the variable index set to each value between start and finish, inclusive. Thus, you can use a for loop to have the variable i count from 1 to 100 like this:

```javascript
for (var i = 1; i <= 100; i++)
```

Each of these patterns includes a new operator written as a double-plus sign. The ++ operator is shorthand for adding one to the variable that precedes it, which is called incrementing the variable. Symmetrically, JavaScript also defines the operator --, which decrements the variable by subtracting one from its previous value.

The increment and decrement operators are a standard feature in a family of programming languages that includes C, C++, Java, and JavaScript—all of which are derived from the C programming language developed at AT&T Bell Labs around 1970. Moreover, if you study any of these languages in detail, you will discover that the complete definition of ++ and -- is considerably more complex than the simple explanation used in this book. The simple forms, however, are sufficient for the idiomatic for loop patterns in which they typically appear.

Although the two for loop patterns described earlier in this section represent the most common uses, the for loop is considerably more general than these patterns suggest. The general form of the for loop in JavaScript looks like this:

```javascript
for (init; test; step) {
    statements
}
```

This code is equivalent to the following while statement:

```javascript
init;
while (test) {
    statements
    step;
}
```

The code fragment specified by init, which is typically a variable declaration, runs before the loop begins and is most often used to initialize an index variable. The test expression is a conditional test written just as it is in a while statement. As long as the test expression is true, the loop continues. The step expression indicates how the value of the index variable changes from cycle to cycle. The most common form of step specification is to increment the index variable using the ++ operator, but this is not the only possibility. For example, you can use the -- operator to write a for loop that counts backward.
You can use the `for` loop to write a function that calculates the factorial of a number $n$—usually written as $n!$—which is defined to be the product of all integers between 1 and $n$. The code for a function `fact(n)` that calculates the factorial of $n$ looks like this:

```javascript
function fact(n) {
    var result = 1;
    for (var i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

If you haven’t done any programming before, it is useful to go through the steps of the `fact` function for a specific example. Suppose that you type `fact(6)` into the console window after defining this function. JavaScript evaluates the function by copying the value 6 into the parameter variable $n$ and then executes the statements in the body of the function. The first statement declares a new local variable called `result` and assigns it the initial value 1. The `for` loop then executes the statements in its body for each value with the index variable $i$ counting from 1 to the value on $n$. On each cycle of the loop, the shorthand assignment statement

```javascript
result *= i;
```

multiplies the value of `result` by the loop index $i$. The final value returned to the caller is therefore $1 \times 2 \times 3 \times 4 \times 5 \times 6$, which is 720.

### 2.5 Finding the greatest common divisor

Another mathematical algorithm—and one that is still in use today—is named for the Greek mathematician Euclid, who lived in Alexandria during the reign of Ptolemy I (323–283 BCE). In his great mathematical treatise entitled *Elements*, Euclid outlines a procedure for finding the greatest common divisor (or gcd for short) of two integers $x$ and $y$, which is defined to be the largest integer that divides evenly into both. For example, the greatest common divisor of 49 and 35 is 7 because 7 is the largest whole number that divides evenly into both 49 and 35.

In modern English, Euclid’s algorithm can be described as follows:

1. If $x$ is divisible by $y$, stop and return $y$ as the result.
2. Store the remainder of $x$ and $y$ in a new variable named $r$.
3. Set $x$ equal to the old value of $y$, set $y$ equal to $r$, and repeat the entire process.
It is easy to translate this process into the following function definition:

```javascript
function gcd(x, y) {
    while (x % y !== 0) {
        var r = x % y;
        x = y;
        y = r;
    }
    return y;
}
```

Figure 2-3 illustrates Euclid’s algorithm in a form that adopts the classical practice of interpreting numbers as distances. For example, when Euclid set out to find the greatest common divisor of two numbers, such as 714 and 210, he framed the problem as one of finding the longest measuring stick that could be used to mark off each of the two distances. The remainder operation is then equivalent to marking off one stick with multiple copies of a second. For example, step 2 in Figure 2-3 shows that the remainder of 714 divided by 210 is 84.

**Figure 2-3** Steps in calculating the greatest common divisor of 714 and 210

Step 1. Lay out the values 714 and 210 as distances x and y.

<table>
<thead>
<tr>
<th></th>
<th>714</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>210</td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

Step 2. Measure x in multiples of y, leaving 84 as the remainder.

<table>
<thead>
<tr>
<th></th>
<th>714</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>210</td>
</tr>
<tr>
<td>y</td>
<td>210</td>
</tr>
</tbody>
</table>

Step 3. Set x to 210, y to 84, and repeat the remainder operation.

<table>
<thead>
<tr>
<th></th>
<th>210</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>84</td>
</tr>
</tbody>
</table>

Step 4. Set x to 84, y to 42, and repeat the remainder operation.

<table>
<thead>
<tr>
<th></th>
<th>84</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>42</td>
</tr>
</tbody>
</table>

Step 5. The remainder is 0, so verify that 42 is the greatest common divisor.

<table>
<thead>
<tr>
<th></th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>
Euclid’s algorithm is considerably more efficient than any strategy you would be likely to discover on your own, and is still used today in a variety of practical applications, including the implementation of the cryptographic protocols that enable secure communication. At the same time, it is not easy to see exactly why the algorithm gives the correct result. Fortunately, Euclid was able to prove the correctness of his algorithm in \textit{Elements}, Book VII, proposition 2. Having a formal proof of the correctness of the algorithm increase one’s confidence in the correctness of the implementation.

2.6 Checking for primes

Another mathematical problem that dates back to antiquity is that of determining whether a given number is prime. A positive integer \( n \) is \textit{prime} if it has exactly two positive divisors, which are always itself and 1. For example, 23 is prime because there are no numbers except 1 and 23 that divide it evenly. The number 35, on the other hand, is not prime because, in addition to the factors 1 and 35, is also divisible by 7 and 5. Integers like 35 that have more than two factors are said to be \textit{composite}. By definition, the integer 1 is neither prime nor composite, because it has only one divisor.

As a programmer, how would you go about designing a function to determine whether an integer \( n \) is prime? If you work directly from the definition, the most obvious approach would be to count the number of divisors and see if there are exactly two. Common sense indicates that any divisors of \( n \) must be less than or equal to \( n \), so if you check all integers between 1 and \( n \), you will find every divisor. This observation suggests that you can determine whether \( n \) is prime by following these steps:

1. Check each number between 1 and \( n \) to see whether it divides evenly into \( n \).
2. Add 1 to a running count each time you encounter a new divisor.
3. Check to see whether the divisor count is 2 after all numbers have been tested.

You can use this strategy as the basis for the implementation of a function called \texttt{isPrime} that tests whether a number is prime, which looks like this:

```javascript
function isPrime(n) {
    var divisors = 0;
    for (var i = 1; i <= n; i++) {
        if (n % i === 0) divisors++;
    }
    return divisors === 2;
}
```
The implementation uses the variable *divisors* to keep track of the number of divisors found so far. At the beginning of the program, *divisors* is set to 0 and is incremented every time a new divisor is found. The number *n* is prime if the count of the divisors is exactly two after all numbers between 1 and *n* have been tested. The test is written using the conditional expression *divisors* == 2, and the function returns the value of that expression as its result. The function *isPrime* thus returns either *true* or *false*. Functions that return Boolean values play an important role in programming and are called *predicate functions*.

The strategy used in this implementation of *isPrime* is not particularly clever or efficient, but it does have one highly desirable property. It works. The *isPrime* function represents an algorithm for determining whether a number is prime. To demonstrate that *isPrime* is indeed an algorithm, it helps to recall the requirements imposed on algorithms, as they were presented at the beginning of this chapter.

The first criterion for an algorithm is that it must be expressed in a form that is clear and unambiguous? For algorithms expressed in English, this condition is difficult to meet. Like all other human languages, English can be fuzzy. When you try to express an algorithm in English, you are likely to leave out a step or gloss over some critical detail. When you express an algorithm in a programming language, the language definition specifies a precise interpretation.

The second requirement for an algorithm is that it must be effective in the sense that it is possible to carry out the individual steps. Once again, the fact that the algorithm has been presented in the form of a program helps to meet this criterion. The JavaScript programming language assigns meaning to each of the constructs in the program, which the JavaScript interpreter can perform.

The third requirement is that an algorithm must terminate. You can see that this criterion is satisfied by looking at the code. The only long-running part of the function is the *for* loop, which goes through exactly *n* cycles each time *isPrime* is called. If *n* is large, the function may take a long time, but it must return eventually.

Greek mathematicians took a special interest in *perfect numbers*, which are numbers that are equal to the sum of all divisors less than the number itself. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14. Using *isPrime(n)* as a model, write a predicate method *isPerfect(n)* that returns *true* if *n* is perfect, and *false* otherwise.
2.7 Library functions

Like all modern languages, JavaScript makes several repositories of useful definitions available to programmers for use as tools. Generically, such repositories are called libraries. In JavaScript, most libraries are implemented as part of a class, which you can think of for the moment simply as a structure that unifies a related set of definitions. Figure 2-4, for example, lists several constants and functions available in JavaScript’s Math class.

**Figure 2-4** Selected definitions from the JavaScript Math library

<table>
<thead>
<tr>
<th>Mathematical constants</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.PI</td>
<td>The mathematical constant π.</td>
</tr>
<tr>
<td>Math.E</td>
<td>The mathematical constant e, which is the basis for natural logarithms.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>General mathematical functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.abs(x)</td>
<td>Returns the absolute value of x.</td>
</tr>
<tr>
<td>Math.min(x, y)</td>
<td>Returns the smaller of x and y.</td>
</tr>
<tr>
<td>Math.max(x, y, ...)</td>
<td>Returns the largest of the arguments.</td>
</tr>
<tr>
<td>Math.min(x, y, ...)</td>
<td>Returns the smallest of the arguments.</td>
</tr>
<tr>
<td>Math.round(x)</td>
<td>Returns the closest integer to x.</td>
</tr>
<tr>
<td>Math.floor(x)</td>
<td>Returns the largest integer less than or equal to x.</td>
</tr>
<tr>
<td>Math.ceil(x)</td>
<td>Returns the smallest integer greater than or equal to x.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logarithmic and exponential functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.exp(x)</td>
<td>Returns the exponential function of x (e^x).</td>
</tr>
<tr>
<td>Math.log(x)</td>
<td>Returns the natural logarithm (base e) of x.</td>
</tr>
<tr>
<td>Math.pow(x, y)</td>
<td>Returns x^y.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trigonometric functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.cos(theta)</td>
<td>Returns the cosine of the radian angle theta.</td>
</tr>
<tr>
<td>Math.sin(theta)</td>
<td>Returns the sine of the radian angle theta.</td>
</tr>
<tr>
<td>Math.tan(theta)</td>
<td>Returns the tangent of the radian angle theta.</td>
</tr>
<tr>
<td>Math.atan(x)</td>
<td>Returns the principal arctangent of x, which lies between −π/2 and +π/2.</td>
</tr>
<tr>
<td>Math.atan2(y, x)</td>
<td>Returns the angle between the x-axis and the line through the point (x, y).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Random number generator</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.random()</td>
<td>Returns a random number that is at least 0 but strictly less than 1.</td>
</tr>
</tbody>
</table>
In JavaScript, you can use the facilities available in a class by writing the class name, a dot, and the name of the constant of function you want to use. For example, the expression `Math.PI` represents the constant named `PI` in the `Math` class, which is defined to be as close an approximation as possible to the mathematical constant π. Similarly, the function call `Math.sqrt(2)` returns the best possible approximation of the square root of 2. It is generally preferable to use library functions when they exist, but it is still good to know that you could define these functions on your own.

Another library class that is particularly useful in the SJS implementation of JavaScript is the `Console` class, which defines a function called `println`. The `Console.println` function makes it possible to display a line on the console. The argument to the function is a JavaScript value called a `string`, which is conceptually a sequence of characters. You will learn more about strings in Chapter 4, but for now all you need to know is that you can create a string by enclosing characters in quotation marks and, moreover, can combine strings and numbers together using the `+` operator. For example, the following function displays a table of the factorials between the two arguments you specify:

```javascript
function printFactorialTable(low, high) {
    for (var i = low; i <= high; i++) {
        Console.println(i + "! = " + fact(i));
    }
}
```

Together with the `fact` function from page 43, you could use this function to create the following console script:

```
JavaScript Console
> printFactorialTable(0, 7)
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
```

Write a program that uses the `isPerfect` function from the puzzle box on page 46 that displays on the console all perfect numbers in the range between 1 and 10000. The first two perfect numbers should be 6 and 28. Your program should find two other perfect numbers in that range as well.
CHAPTER 3

Babbage Machines

What would the world have been like if computers had been invented a century earlier? That question—which is by no means as speculative as it sounds—forms the basis for a 1991 science-fiction novel by William Gibson and Bruce Sterling. The title of their novel is *The Difference Engine*, which is also the name of a sophisticated mathematical calculator conceived in the 1820s. Although the original Difference Engine was never completed, its design represented an enormous intellectual leap forward that anticipated many aspects of modern computers.

The Difference Engine—along with its unrealized but vastly more powerful successor, the Analytical Engine—was the brainchild of Charles Babbage, Fellow of the Royal Society and Lucasian Professorship of Mathematics, the prestigious chair at Cambridge University once held by Isaac Newton and more recently occupied by Stephen Hawking. In the early 19th century, Babbage became convinced that the many errors present in the mathematical tables of his day could be corrected only by producing such tables through a mechanical process insusceptible to the vagaries of human calculation. In his 1864 memoir entitled *Passages from the Life of a Philosopher*, Babbage recalls that his interest in mechanical computation began sometime in 1812 or 1813:

One evening, I was sitting in the rooms of the Analytical Society, at Cambridge, my head leaning forward on the Table in a kind of dreamy mood, with a Table of logarithms lying open before me. Another member, coming into the room, and seeing me half asleep, called out, “Well, Babbage, what are you dreaming about?” to which I replied, “I am thinking that all these Tables (pointing to the logarithms) might be calculated by machinery.”
Ada Augusta Lovelace and the invention of programming

The Analytical Engine that Charles Babbage designed anticipated many features of modern computing systems. Most importantly, the Analytical Engine was intended to serve as a general-purpose machine, capable of solving many different kinds of problems. For such a machine to work, there must be some way to specify what problem the machine is supposed to solve at any given time. In today’s world, that process is called programming.

In the eyes of many historians, the genesis of programming as an abstract idea is due less to Babbage than it is to Ada Augusta Lovelace, the daughter of Lord Byron and Annabella Milbanke. Ada was fascinated by mathematics and science, even though she never had an opportunity for formal study, given the expectations for women in the early 19th century. She therefore read extensively on her own and corresponded with leading scientists.

The young Ada Byron met Babbage in 1833 when she and her mother attended one of Babbage’s noted Saturday night soirées in London. On display at that gathering was a model of the Difference Engine. More than most of Babbage’s guests, Ada immediately recognized the power of the Difference Engine, which she described as a “thinking machine.”

For the remaining twenty years of her all-too-short life, Ada maintained a close, collaborative friendship with Babbage and remained one of his strongest supporters. When the Italian engineer Luigi Menabrea published a report on the Analytical Engine in 1842, Babbage encouraged Ada to translate it. In doing so, Ada extended the work substantially, adding translator’s notes nearly three times as long as the paper.

It is clear from her correspondence that Ada appreciated the potential of the Analytical Engine even more than Babbage, particularly in her recognition that the machine could manipulate data other than numbers. Her contributions to computing were given further recognition in 1984 when the programming language Ada was named in her honor.

Ada Augusta Lovelace’s program for solving a pair of first-degree equations (Note D from her translation of Menabrea, 1843)

<table>
<thead>
<tr>
<th>Number of Operations</th>
<th>Variables for Data.</th>
<th>Working Variables.</th>
<th>Variables for Results.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>2</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>8</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>9</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>10</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
<tr>
<td>11</td>
<td>m</td>
<td>n</td>
<td>d</td>
</tr>
</tbody>
</table>

The table above shows Ada Augusta Lovelace’s program for solving a pair of first-degree equations. The variables for data include m, n, d, m', n', d', m'', n'', d''. The working variables include m''', n''', d'''. The variables for results include m'''', n'''', d'''', where m''' = x, n''' = y.
Errors in published tables had considerable economic consequence in Babbage’s
time. In the absence of the calculators that we use so effortlessly today, anyone in
the early 19th century who worked in quantitative fields—scientists, engineers,
navigators, surveyors, businessmen, and many others—had to rely on vast
compendia of mathematical tables whose construction required significant human
labor and the consequent expense. These tables often contained serious errors that
would invalidate any computations derived from the incorrect data. Over time,
some of these errors would be discovered and corrected, leading to the publication
of a succession of reports listing the errors in previously published works. These
new calculations, of course, were also susceptible to error, as Babbage observes in
his memoir:

In 1828 I lent the Government an original MS. of the table of
Logarithmic Sines, Cosines, &c., computed to every second of the
quadrant, in order that they might have it compared with Taylor’s
Logarithms, 4to., 1792, of which they possessed a considerable number
of copies. Nineteen errors were thus detected, and a list of these errata
was published in the Nautical Almanac for 1832: these may be called

Nineena errata of the first order . . . . . 1832

An error being detected in one of these errata, in the following Nautical
Almanac we find an

Erratum of the errata in N. Alm. 1832 . . . . . 1833

But in this very erratum of the second order a new mistake was
introduced larger than any of the original mistakes. In the year next
following there ought to have been found

Erratum in the erratum of the errata in N. Alm. 1832 . . 1834

Surely not a happy state of affairs.

3.1 The Difference Engine

Babbage’s belief that mechanical computation was the key to solving the “tables
crisis” led him to develop plans for a new kind of computing machine, which he
began to pursue seriously toward the end of 1821. Babbage called his machine the
Difference Engine because its internal operation relied on calculating differences
between terms in mathematical sequences.

The basic principle of computing by extracting differences is not at all difficult
to understand. In his memoir, Babbage offered a straightforward explanation of the
general idea, which appears in Figure 3-1. The fundamental insight is that
successive terms in a tabular series often differ from each other in easily calculable
ways. In Babbage’s first example, the cost of each additional pound of meat is
Explanation of the Difference Engine.

Those who are only familiar with ordinary arithmetic may, by following out with the pen some of the examples which will be given, easily make themselves acquainted with the simple principles on which the Difference Engine acts.

It is necessary to state distinctly at the outset that the Difference Engine is not intended to answer special questions. Its object is to calculate and print a series of results formed according to such laws. These are called Tables—many such are in use in various trades. For example, there are collections of Tables of the amount of any number of pounds from 1 to 100 lbs. of butchers’ meat at various prices per lb. Let us examine one of these Tables: viz.—the price of meat at 5d. per lb., we find

<table>
<thead>
<tr>
<th>Number</th>
<th>Lbs.</th>
<th>Price.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

There are two ways of computing this Table:

1st. We might have multiplied the number of lbs. in each line by 5, the price per lb., and have put down the result in l. s. d., as in the second column: or,

2nd. We might have put down the price of 1 lb., which is 5d., and have added five pence for each succeeding lb.

Let us now examine the relative advantages of each plan. We shall find that if we had multiplied each number of lbs. in the Table by 5, and put down the resulting amount, then every number in the Table would have been computed independently. If, therefore, an error had been committed, it would not have affected any but the single tabular number at which it had been made. On the other hand, if a single error had occurred in the system of computing by adding five at each step, any such error would have rendered the whole of the rest of the Table untrue.

Thus, the system of calculating by differences, which is the easiest, is much more liable to error. It has, on the other hand, this great advantage: viz., that when the Table has been so computed, if we calculate the last term directly, and if it agree with the last term found by continual addition of 5, we shall then be quite certain that every term throughout is correct. In the system of computing each term directly, we possess no such check upon our accuracy.
...fivepence, which means that you can construct a table showing the price of various quantities of meat using simple addition. In the butcher-shop example, the differences between successive entries—which are called **first differences** in the discussion—are constant, which makes for a particularly easy calculation. In his second example of marbles arranged in a triangular pattern, the difference between successive entries now changes, but in a predictable way. In this case, it is the differences between the differences—which are called **second differences**—that remain constant. For any mathematical function that can be represented as a polynomial, calculating successive differences will eventually yield a constant.

The mathematical principle of extracting differences was well known in Babbage’s day and had indeed been central to the development of calculus by Newton and Leibniz more than a century before. What Babbage sought to do was to use this principle to build a physical machine capable of carrying of computing terms in a tabular series by, in essence, reversing the process of extracting differences. If successive terms in a series differ by a constant value, one can compute each new value by adding that constant difference to a running total.

By the middle of 1822, Babbage had created a physical model of a small Difference Engine capable of producing—in the words of his letter to Sir Humphry Davy, President of the Royal Society, on July 3—“any tables whose second differences are constant.” His brief description contained in that letter suggests that Babbage’s prototype was functionally similar to the three-column Difference Engine model shown in Figure 3-2, which was built a decade later and now resides in the Science Museum of London. Babbage’s purpose in writing the letter was to seek government support for the completion of a much larger machine, capable of maintaining five orders of differences with twelve decimal digits in each value. Babbage’s design also included a mechanical printer connected directly to the Difference Engine to eliminate the possibility of transcription and typesetting errors.

The photograph in Figure 3-2 provides a sense of the appearance of a small Difference Engine but gives little insight into its workings, which require a bit of explanation. Numbers in the Difference Engine are stored in vertical columns, as illustrated by the diagram on the right. The units digit of each number appears at the bottom of the column, which means that numbers read downward. The number shown in the diagram is therefore 1792, the year of Babbage’s birth.

The three columns in the prototype machine—and the much larger number of columns available in the Difference Engine that Babbage sought to build—keep track of the differences. In the prototype, the leftmost column represents the output and displays the values to be recorded. The middle column holds the first difference, and the right column holds the second difference. In a larger machine, each successive column would hold the difference of the next higher order.
FIGURE 3-2 Photograph of Babbage’s three-column model of the Difference Engine (1832)
Although Babbage envisioned that the Difference Engine would eventually be powered by steam, the prototype required the operator to turn the crank shown at the top of the photograph. Turning the crank advanced the machine through one cycle of its operation, which, at least in the case of the three-column prototype, had the following effect:

1. The value in the middle column was added to the value in the left column to produce the next output value.
2. The value in the right column was added to the value in the middle column to produce the appropriate first difference for the next cycle.

To see how this process works, it is useful to consider a specific example. Suppose that the Difference Engine has been set by hand so that the columns from left to right contain the values 0, 1, and 2. Turning the crank adds 1 to the 0 in the left column and then goes back and adds 2 to the center column, as illustrated by the following diagram, where the numbers at the bottom are included just to make the value in each column easier to read:

In his demonstration machine, Babbage added a set of wheels at the upper right to keep track of the cycle count, which indicates how many times the handle has been turned. In the preceding diagram, for example, turning the crank advances the cycle count from 0 to 1, as shown in the following close-up view:
After the first cycle, turning the crank again goes through the same operations. The 3 in the middle column is added to the 1 in the left column to produce the new output value 4. As part of that same operation, the 2 in the right column is added to the 3 in the middle column to produce the value 5. The state of the machine after two cycles therefore looks like this:

![Diagram of the machine](image)

The numbers displayed in the left column for the three machine configurations so far are 0, 1, and 4. If you continue this process, the output values form the sequence 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, and so on.

From this simple analysis, it seems clear that the Difference Engine is generating a table of squares. To see why, it helps to construct a table of differences, just as Babbage did in his explanation of the Difference Engine in Figure 3-1. For the sequence of squares, Babbage’s table looks like this:

<table>
<thead>
<tr>
<th></th>
<th>Table.</th>
<th>1st Difference.</th>
<th>2nd Difference.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first line in the table—0, 1, and 2—simply records the initial column values used for this example. The magic of calculating by differences does the rest.
Babbage was able to secure government funding to build the Difference Engine even though public funding of science and technology was not a common practice in the 19th century. Over the next decade, Babbage received a total of £17,470 from the British government in support of his project, a very substantial sum for that time. Unfortunately, that investment never produced a full-scale model, and the project was abandoned in 1834.

Historians of technology have offered many explanations for Babbage’s failure to complete the Difference Engine. Some have suggested that the precision and scale of Babbage’s vision were beyond the capacity of mechanical engineering at the time. This explanation, however, is undermined by the fact that in 1843 the Swedish father-and-son team of Georg and Edvard Scheutz were able to construct a working Difference Engine of a relatively modest scale from reading a general description of Babbage’s plan. For the 1991 bicentennial of Babbage’s birth, the Science Museum in London built a full-scale Difference Engine from Babbage’s original plans. That experience convinced the builders not only that Babbage’s design would work but also that there would have been no fundamental technical barriers to completing the project in Babbage’s day.

A more likely explanation of Babbage’s failure to bring the Difference Engine to completion is simply that, in the process of working on it, he became convinced that a far more effective computational engine was possible. His new vision went far beyond the ideas in the Difference Engine to the point that pursuing the older design must have seemed pointless given the prospect of a dramatically more powerful computing machine that Babbage called the Analytical Engine. Babbage began to design that machine in 1834 and continued the project, at his own expense, throughout most of the rest of his life.

In a way, Babbage’s experience with the Differential Engine and its unrealized successor offers an early example of an important principle of system development identified by Fred Brooks in his landmark collection of essays entitled The Mythical Man Month. In Chapter 11, Brooks notes that it always pays to build a prototype.

To make sure that you understand how the machine works, try to figure out the initial conditions of the Difference Engine needed to generate each of the following sequences:

(a) The even integers: 0, 2, 4, 6, 8, 10, 12, 14, 16, . . .
(b) The sum of the first $N$ even integers: 0, 2, 6, 12, 20, 30, 42, 56, . . .
(c) The cubes of the integers: 0, 1, 8, 27, 64, 125, 216, . . .

The last example requires a Difference Engine with four columns.
system because the process of developing that prototype makes it clear how the system should have been designed in the first place:

Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers.

Unfortunately, Brooks’s insights into the process of system design came a century and a half too late for Babbage. His backers in government and the scientific establishment did not relish the prospect of throwing away the £17,470 they had invested in the Difference Engine. Dismayed by that experience, his earlier supporters could not bring themselves to finance the development of the Analytical Engine that was in some sense the most important product of that investment.

3.2 The Analytical Engine

The fundamental feature that separates Babbage’s new design from its predecessor is the Analytical Engine was designed to be programmable. Rather than running through a fixed computational process based on the initial settings of the numeric columns, the Analytical Engine could perform the fundamental arithmetic operations of addition, subtraction, multiplication, and division in a sequence specific to the problem at hand. The program itself was encoded in a set of punched cards that the Analytical Engine would read as it executed its operations.

The concept of controlling a mechanical process through punched cards did not originate with Babbage. The idea was already in widespread use in a mechanical loom developed by the French inventor Joseph-Marie Jacquard, based on earlier designs by Jacques de Vaucanson and Basile Bouchon. In Jacquard’s machine, the presence or absence of a hole at a specific place in a card would determine whether a weaving shuttle passed above or below the background mesh, thereby controlling the pattern on the cloth. The original version of the picture of Jacquard shown in the margin is in fact woven in silk at a level of detail comparable to that of a modern laser printer.

In addition to revolutionizing the process of weaving, Jacquard’s invention also provided Babbage with a model for a programmable machine. Babbage recognized that the holes in punched cards could also be used to represent numbers, as well as coded representations for the internal operations of the machine. These concepts represent a profound intellectual advance over the design of the Difference Engine and enabled Babbage to anticipate many of the features of modern computers.
One of the most important ideas that Babbage incorporated into his design was the separation of the machine into two functional components, one to store the data and another to perform calculations. In his memoir, Babbage described that division as follows:

The Analytical Engine consists of two parts:—
1st. The store in which all the variables to be operated upon, as well as all those quantities which have arisen from the result of other operations, are placed.
2nd. The mill into which the quantities about to be operated upon are always brought.

As you will learn in Chapter 6, modern computers maintain much the same division of functionality, even though they are called by different names.

Explaining the operation of the Analytical Engine is not an easy task, mostly because there is considerable uncertainty about its internal details. For the Difference Engine, Babbage left complete plans that were sufficiently precise for the Science Museum to build a replica. In the case of the Analytical Engine, Babbage’s notes are far more general. To make matters worse, Babbage proposed several different designs for the Analytical Engine at different times in his life. These designs share certain broad principles but vary substantially in their details. The most complete account was in fact not written by Babbage at all, but by an Italian military engineer named Luigi Menabrea who attended a set of lectures that Babbage gave in Turin on the machine. Menabrea’s notes—translated and extensively annotated by Ada Lovelace—provide the best guide to the structure of the machine, but even these notes leave many details unspecified.

Given these many uncertainties, developing a simulator for the Analytical Engine involves a certain amount of speculation. The design outlined in the rest of this chapter is adapted from John Walker’s simulation of the Analytical Engine at Fourmilab in Switzerland, which is itself synthesized from the various published descriptions.

For both the mill and the store, Babbage envisioned a mechanical representation of numeric values similar to that used in the Difference Engine. In particular, numbers were represented using columns of individual digits, just as they were for the earlier design. For the Analytical Engine, however, Babbage felt the need for a dramatic expansion in scale. In his written accounts of the Analytical Engine, Babbage envisions a machine with 1000 columns of figures, each of which holds a 50-digit number. Because a machine of that size—more than 1000 times larger than the prototype Difference Engine that Babbage actually built—is impossible to represent on the page, the diagrams in this chapter show a smaller machine in which there are only 15 columns, each of which contains a seven-digit number. The structure of this machine appears in Figure 3-3.
Note that each column—which Babbage called a variable—is marked along the top with a subscripted name: \(v_0\), \(v_1\), \(v_2\), and so on. These subscripts provide a way of specifying which variables in memory are involved in a computation. Since the subscript is numeric, it can be represented on the punched cards controlling the operation of the machine, just as the numbers used to represent data values are. This numeric specification of a variable in memory is called an address. An instruction for the Analytical Engine typically consists of an instruction code followed by the address on which that instruction operates, as illustrated in the following section.

The structure of the mill is more complicated, although it also uses mechanical columns to represent numeric values. The columns labeled \(I_1\) and \(I_2\) represent what Babbage called the ingress columns and hold the inputs to an arithmetic operation. The column labeled \(E\) is the egress column and contains the result after an arithmetic operation is performed. (The columns \(I_1'\) and \(E'\) and the boxes marked \(op\) and \(runup\) are discussed later in this chapter.)

The operation of the mill differs in several important respects from that of modern electronic computers, although most of the differences make sense in the context of a mechanical machine. Babbage wanted the Analytical Engine to be able to add, subtract, multiply, and divide, but the mechanism used to implement each of these operations is different. Setting the machine up to perform a new operation is very costly in time. Babbage sought to minimize this cost by designing the machine so that one could perform, for example, a series of additions followed by a series of multiplications with only a single change in the operational configuration of the machine. Thus, instead of envisioning a \(+\) operation that takes two values and computes their sum, Babbage chose to have the \(+\) operation simply set the machine
up for doing addition. Once the configuration is set, the programmer can feed pairs of values to the mill, which the Analytical Engine then dutifully adds together.

**Specifying operations for the Analytical Engine**

Babbage’s descriptions of the Analytical Engine make it clear that he envisioned that the instructions for the machine would be punched on cards in a fashion similar to that of the Jacquard loom. For example, in Chapter 8 of *Passages from the Life of a Philosopher*, Babbage suggests that the cards representing the four arithmetic operations would be punched as follows:

![Diagram of operation cards]

The specific encoding of each operation as a set of holes punched onto a card is not important in terms of understanding the operation of the Analytical Engine. What is important is that you recognize that it is possible to design an encoding that allows the machine to perform the correct operations. In the examples in this chapter, the instructions are given symbolically, usually as a single character. Thus, four of the instructions for the Analytical Engine are simply the standard symbols for the arithmetic operations: +, −, ×, and ÷. However, because the multiplication and division symbols, however, don’t appear on a standard keyboard, the simulator for the Analytical Engine requires you to type these symbols in their standard programming form, using * for multiplication and / for division. To make it easier for you to keep track of the current operation, the mill diagrams in this chapter show the current operation in the box marked op.

Babbage in fact anticipated that the Analytical Engine would have three different types of instruction cards, each of which would be read by a separate reader. The four arithmetic operators were described as **operation cards** and had the format suggested in the diagram. The machine also supported **variable cards**, which were used to specify the transfer of information from a specified variable in the store to the computational units in the mill, or vice versa. Finally, Babbage envisioned a set of **number cards**, whose purpose is to provide initial values to the variables in the store. To simplify the structure of the machine and make its operation easier to understand, the Analytical Engine simulator that accompanies this book assumes that all these cards are combined together into a single set of instructions that collectively represents the **program** for the machine.

Before you can write a complete program for the Analytical Engine, you need to understand a bit more about how it performs computation and the pattern of interaction between the mill and the store. The details of the machine operation are outlined in the next few sections.
Specifying data for computation

As noted earlier in the chapter, the operation cards (+, −, x, and ÷) don’t actually trigger any computation, but instead set up the machine so that the mill is ready to perform that operation when it receives the necessary data. The values for any computation come from the variables in the store, which must be copied into the mill for processing. Similarly, once the result has been obtained, the program must copy it back into a variable for later reference. Moving data back and forth between the mill and the store requires a pair of additional instructions called **load** and **store**, which are represented by the single letters L and S, respectively. Both instructions require an address to indicate which of the variables in the store is involved. For example, the instruction that loads the value from variable v₁₃ into the mill is represented like this:

\[ L \ 13 \]

Arithmetic operations in the Analytical Engine always consist of a pair of load instructions. The first load instruction has the effect of copying the value from the specified address into the mill using ingress column I₁. The next load instruction copies the requested value into ingress column I₂ and then immediately performs the current operation on the values in the two ingress columns, placing the result of that operation on egress column E. The contents of the two ingress columns are then reset so that the next load instruction transfers its value to ingress column I₁, as in the original configuration.

The effect of this design means that applying an arithmetic operation typically requires the following four-step process:

1. Set up the machine for the desired mathematical operation (+, −, x, or ÷) using the corresponding operation card. If several operations of the same type occur in sequence, this step is required only once at the beginning.
2. Load the first value into ingress column I₁ by using the L instruction in conjunction with the address of the column that contains the data.
3. Load the second value from memory using an L instruction specified exactly as in the previous step. Because this is the second such operation, the Analytical Engine copies the value into ingress column I₂ and initiates the current arithmetic operation, placing the result on the egress column E.
4. Store the result from the egress column back into memory using an S instruction with the address of the variable designated to receive the result.

This sequence of operations is best illustrated by a simple example. Suppose that you want the Analytical Engine to add the values in memory variables v₁₂ and
$v_{13}$ and then store the result back in $v_{12}$. This calculation can be performed by the following set of instruction cards:

+  
L 12  
L 13  
S 12

The first operation sets the machine up for addition, the next two indicate the location of the data, and the final operation stores the result back in memory.

Babbage, however, was not content with a computational engine that would merely store its results in memory. To address the crisis of correctness in published tables, Babbage insisted that the Analytical Engine be connected to a printer that could record these results on paper. To realize this aspect of the machine’s design, the simulator for the Analytical Engine includes a command, indicated by the letter $P$, to print the value from a memory cell. For example, to print out the sum stored in $v_{12}$, you would simply add the line

$P 12$

to the preceding program.

As a second example, suppose that you wanted to subtract 5 from the number stored in memory variable $v_{10}$. The general structure of this operation is similar, but there is a small wrinkle: before you can subtract the value 5, you have to make sure that the value 5 is stored in one of the memory variables. To do so, you need to include a number card in your program that specifies the variable to use for the desired value. If, for example, you wanted to store the value 5 in $v_{11}$, you could do so by including the card

$N 11 5$

Once you have ensured that $v_{11}$ contains the value 5, the rest of the program to subtract 5 from variable $v_{10}$ is straightforward:

-  
L 10  
L 11  
S 10

Explicit values like the number 5 used in this example are called **constants**. Constants occur frequently in programming and are represented in modern computers by initializing a memory location to hold the constant value—precisely the strategy that Babbage suggested more than a century and a half ago.
**Multiplication and division**

Although addition and subtraction for the Analytical Engine can be implemented using the same mechanical logic as in the Difference Engine, multiplication and division are more complex and require considerably more time. Babbage estimated that the Analytical Engine could execute an addition or subtraction instruction in a second, but concluded that multiplying two 50-digit numbers would take at least a minute. Multiplication, however, raises another interesting problem. If you add two 50-digit numbers, the result is also 50-digit number unless the values are so large that there is a carry into the 51st place, which is presumably not a common occurrence. If, however, you multiply two 50-digit numbers, the result can have as many as 100 digits. Babbage anticipated this problem and included a second egress column $E'$ to support the larger values that arise in multiplication. When the mill multiplies two numbers, the result appears in both the $E'$ and $E$ columns, which together represent a 100-digit number. The first half of the digits appear in the $E'$ column, and the second half appear in the $E$ column. As long as the numbers are small, however, the value in the $E'$ column will be 0 and can therefore be ignored.

Division presents a similar situation. Just as the product of two 50-digit numbers might require 100 digits, Babbage reasoned that the first operand in a division operation might also be that large. To accommodate these larger values, Babbage included an additional $I'_1$ column whose only purpose is to hold the first half of the digits involved in a division operation. Once again, this value will be zero unless large numbers are involved, which means that you can ignore the $I'_1$ column in most applications.

Division is also complicated by the fact that it in some sense produces two results: the **quotient**, which is the number of times that the divisor goes into the dividend, and the **remainder**, which represents the value left over after the division is complete. For example, if you divide 21 by 4, the quotient is 5 and the remainder is 1. Babbage wanted to make both values available. In the Analytical Engine simulator supplied with this book, the quotient is stored in the $E'$ column and the remainder simultaneously appears in the $E$ column.

To make it possible to use the primed columns $I'_1$ and $E'$, the Analytical Engine simulator defines the instructions $L'$ and $S'$, which work just like the standard load and store instructions except that they use the appropriate primed column in the mill instead of the standard ingress and egress columns. The $L'$ instruction copies data from the specified variable in memory to the $I'_1$ column, and the $S'$ instruction copies data from the $E'$ column back into the memory.

If the multiplication and division operations seem a little too complicated, you can take solace in the fact that they are not used in the examples or problems in this chapter, which limit themselves to addition and subtraction. My reason for
including them in this discussion is partly to maintain historical accuracy but at least as much to show the extent to which Babbage anticipated problems that would arise more than a century later in the development of modern computing. The strategy of storing the product of two numbers in a pair of memory cells was common practice in electronic computers, as was having the divide operation calculate both the quotient and the remainder. Many of the early computers designed in the middle of the 20th century used essentially the same conventions that Babbage had proposed in his original design.

**Simulating the operation of the Difference Engine**

Suppose that you wanted to use the Analytical Engine to compute the table of squares produced by the Difference Engine earlier in this chapter. Although it would certainly be possible to use the multiplication capabilities the Analytical Engine offers, doing so would slow down the computation considerably. The easiest approach is to employ the method of finite differences used in the earlier example. Like its predecessor, the Analytical Engine uses columns of digits representing integers. If you could arrange to have the first three columns of the Analytical Engine behave in the same way that the columns in the Difference Engine do, you could compute the table of squares essentially by simulating the operation of the Difference Engine.

The first step in the process is to set up the variables $v_0$, $v_1$, and $v_2$ so that they match the initial configuration of the Difference Engine when it is set up to generate squares. That part is easy. All you need to do is use three number cards

| N 0 0 |
| N 1 1 |
| N 2 2 |

to initialize $v_0$, $v_1$, and $v_2$ to 0, 1, and 2, respectively. The idea here is that $v_0$ will hold each new square number as it is generated, $v_1$ holds the first difference (which changes as the computation proceeds), and $v_2$ holds the constant second difference.

When you turn the crank on the three-column Difference Engine, the following actions occur:

1. The value in variable $v_1$ is added to the value in variable $v_0$ to produce the next output value.
2. The value in variable $v_2$ is added to the value in variable $v_1$ to produce the appropriate first difference for the next cycle.

These operations are easy to encode for the Analytical Engine. All you need to do is set up the machine for addition, supply the correct sequence of load and store
instructions to accomplish these two additions, and then print out the value from $v_0$. This task therefore requires the following program:

```
+ 
L 0  
L 1  
S 0  
L 1  
L 2  
S 1  
P 0
```

At the end of these instructions, $v_0$ will contain the value 1 (which will also appear on the printer), $v_1$ will contain 3, and $v_2$ will continue to hold 2, as it will throughout the computation. To generate the next square, all you would need to do is repeat the last seven instructions. You could do so by adding new cards to the program, but there has to be a better way.

**Specifying repetition and conditional execution**

Both Babbage and Lovelace understood that the tabular computations for which the machine was designed would require the Analytical Engine to repeat sequences of operations. Such repetition can be implemented easily by allowing the mechanism that reads the operation cards to “back up” through a set of cards and then execute them again. In her notes on the Menabrea manuscript, Ada Lovelace describes the required operation by appealing to the analogy of the Jacquard loom:

> By the introduction of the system of backing into the Jacquard-loom itself, patterns which should possess symmetry, and follow regular laws of any extent, might be woven by means of comparatively few cards. Those who understand the mechanism of this loom will perceive that the above improvement is easily effected in practice by causing the prism over which the train of pattern-cards is suspended to revolve backwards instead of forwards . . . until, by so doing, any particular card, or set of cards, that has done duty once, and passed on in the ordinary regular succession, is brought back to the position it occupied just before it was used the preceding time. The prism then resumes its forward rotation, and thus brings the card or set of cards in question into play a second time.

To ensure that the Analytical Engine can perform this repetition automatically, it is necessary to expand the legal set of instruction cards to include one that backs up over a certain number of cards. In the simulated Analytical Engine, the backup instruction has the form

```
B n
```
where \( n \) is the number of cards to be shifted backwards through the instruction reader. It also proves useful to be able to skip cards moving forward, which gives rise to a symmetric instruction

\[ F n \]

which skips over the next \( n \) cards without executing them.

Armed with this new capability, it is easy to write a program for the Analytical Engine that prints out an endless table of squares. All you need to do is insert a

\[ B 8 \]

instruction at the end of the program that backs up the card reader to the first load instruction.

The only problem with this program is that it continues to run until someone stops the entire machine. Suppose that you wanted instead to stop the program after it had computed the first 100 squares? How would you program the Analytical Engine to stop at that point?

To solve this problem, it is essential that the program be able to test for certain conditions and to alter its behavior in response. In his *Passages from the Life of a Philosopher*, Babbage makes it clear that the Analytical Engine is indeed capable of responding to conditions, but is frustratingly vague about the details:

> Mechanical means have been provided for backing or advancing the operation cards to any extent. There exist means of expressing the conditions under which these various processes are required to be called into play.

In the absence of more definite guidance from Babbage and his collaborators, computing historians have had to make educated guesses as to what conditions the Analytical Engine could detect. Babbage’s drawings show a set of “Running up levers,” which he uses to indicate that a column has carried out of its range. In the Analytical Engine simulator that accompanies this book, I have followed the lead of Fourmilab’s John Walker, who simplifies the design to have a single *runup lever*, which is a mechanical indicator that can be set when a particular condition occurs. Walker further assumes that the program can check the runup lever and decide whether to change the sequence of instruction cards. In the simulator, you can express this sort of test by writing a ? in front of a \( B \) or \( F \) command, which indicates that the backward or forward motion should occur only if the runup lever is set.

The complete list of instructions for the simulator appears in Figure 3-4. The descriptions of the individual instructions also indicate the conditions under which
**Operation cards for the Analytical Engine**

- **N** \( k \) \( n \) *Number.* Initializes the memory variable \( v_k \) to have the value \( n \). Number cards are used to specify constants used in the calculation and usually appear at the beginning of the program.

- **+** *Add.* Sets the Analytical Engine into addition mode. When the operands are loaded into the ingress column, the two values are added together and placed on the egress column. If the sign of the result differs from that of the value in \( I_1 \) or if the result cannot fit in the number of digits available, the runup lever is set.

- **-** *Subtract.* Sets the Analytical Engine into subtraction mode. When the operands are loaded into the ingress column, the value in \( I_2 \) is subtracted from \( I_1 \) and the result is placed on the egress column. If the sign of the result differs from that of the value in \( I_1 \) or if the result cannot fit in the number of digits available, the runup lever is set.

- **\( \times \) or \( * \)** *Multiply.* Sets the Analytical Engine into multiplication mode. When the operands are loaded into the ingress column, the two values are multiplied together. Because the result of a multiplication is likely to be much larger than the input values, the output is stored in two columns that together act as a column with twice the usual number of digits. The first half of the result is stored in egress column \( E' \) and the second half is stored in egress column \( E \). As long as the numbers are small, you can ignore \( E' \) and take the result from \( E \), just as you would for addition and subtraction. The runup lever is not affected.

- **\( \div \) or \( / \)** *Divide.* Sets the Analytical Engine into division mode. The number contained in the combined columns \( I'_1 \) and \( I_2 \) is divided by the contents of \( I_1 \). The quotient is placed in \( E' \) and the remainder is placed in \( E \). If the quotient does not fit in its column or the divisor is zero, the runup lever is set.

- **L** \( k \) *Load.* Copies the contents of variable \( v_k \) into the appropriate ingress column, which is \( I_1 \) for the first such instruction and \( I_2 \) for the second \( L \) instruction in a pair. The contents of variable \( v_k \) are not affected. If the \( L \) in the instruction is followed by a single quotation mark, the value is copied into \( I'_1 \).

- **S** \( k \) *Store.* Stores the value from the egress column into variable \( v_k \). If the \( S \) is followed by a single quotation mark, the value is taken from \( E' \) instead.

- **P** \( k \) *Print.* Outputs the value from variable \( v_k \) to the printer.

- **F** \( n \) *Forward.* Moves the card reader forward \( n \) cards, skipping those operations. Note that the current card has already been read, so that the instruction \( F \ 1 \) skips the next card.

- **B** \( n \) *Backward.* Moves the card reader backward \( n \) cards, allowing the machine to repeat previous operations. In counting the number of cards to move, it is important to keep in mind that the current operation has already been read.

- **?F** \( n \)
- **?B** \( n \) *Conditional.* These operations are similar to the \( F \) and \( B \) operations except that the card reader moves only if the runup lever is set.
the runup lever is set. For example, when the machine is set for subtraction, the detailed discussion of the operation in Figure 3-4 reveals that the runup lever will be set “if the sign of the result differs from that of the value in I, or if the result cannot fit in the number of digits available.” The first condition is the one that turns out to be more generally useful. If you subtract a number $x$ from a second number $y$, the sign of the result will be different from the sign of $y$ only if $x > y$. This fact means that you can use subtraction and the state of the runup lever to compare two values.

Being able to perform comparisons makes it possible to redesign the table of squares program so that it prints only the first 100 squares and then stops. The code to do so appears in Figure 3-5. Note that this program is annotated with comments using the syntax that you might have seen with languages like Java, C, or C++. As in these other languages, comments do not affect the operation of the machine but instead explain to human readers what the program is doing.

### Figure 3-5 Program for the Analytical Engine to generate a table of squares

```plaintext
/*
 * File: Squares.ae
 * ------------
 * This program generates a table of squares from 1x1 to 100x100.
 */
N 0 0    /* V0 holds the current square */
N 1 1    /* V1 holds the first difference */
N 2 2    /* V2 holds the constant second difference */
N 3 100  /* V3 holds the number of terms left to generate */
N 4 1    /* V4 holds the constant 1 */
-       /* Set the machine for subtraction */
L 3      /* Take the number of terms left */
L 4      /* Subtract one and see if the sign changes */
?F 10    /* If it does, skip forward to the end of the program */
S 3      /* If not, update the number of terms remaining */
+       /* Set the machine for addition */
L 0      /* Load the current square */
L 1      /* Add the first difference */
S 0      /* Store it back as the new square */
L 1      /* Load the first difference */
L 2      /* Update it by adding in the second difference */
S 1      /* And store the value back as the new first difference */
P 0      /* Print out the value of the current square */
B 14     /* Back up to the beginning of the program */
```
Even more than with the Difference Engine, understanding the operation of Babbage’s Analytical Engine requires you to play with it a bit. To test your understanding, try to solve each of the following problems:

(a) Trace the operation of the following program and determine the sequence of values that appear on the printer:

```
N 0 1
N 1 100000
P 0 +
L 0 L 0 S 0 -
L 0 L 1
?B 9
```

What mathematical function is this program computing?

(b) Write a program that displays the cubes of the integer (0, 1, 8, 27, 64, 125, 216, and so on) up to the value of $10^3$.

(c) In the 13th century, the Italian mathematician Leonardo Fibonacci—as a way to explain the geometric growth of a population of rabbits—devised a mathematical sequence that now bears his name. The first two terms in this sequence are 0 and 1, and every subsequent term is the sum of the preceding two. Thus, the first several terms in the Fibonacci sequence look like this:

```
0 1 1 2 3 5 8 13 21 34 55 89
```

Write a program for the Analytical Engine that displays the first 30 terms in the Fibonacci sequence.
Amazing as they were for their time, Babbage’s machines are not the direct ancestors of modern computer hardware. Although those early machines and their modern counterparts share some of the same concepts, modern computers incorporate several new ideas that fundamentally change the way computers are designed. In particular, today’s computers represent information in a simple but powerful form that allows any information—no matter how complex—to be stored as a sequence of primitive values, each of which can be in only one of two possible states. Each of those primitive values is called a bit.

The interpretation of the values for each bit depends to a certain extent on how you choose to view the underlying information. If you think of the bits that form the internal circuitry of the machine as tiny light switches, you might label those states as off and on. If you think of each bit as a logical value, you might instead use the labels false and true. However, because the word bit comes originally from a contraction of binary digit, it is more common to label those states as 0 and 1, which are the digits of the binary number system on which computer arithmetic is based.

The fact that each primitive circuit component has only two possible states gives binary arithmetic a tremendous advantage over the more familiar decimal model. In Babbage’s machines for example, each wheel has ten possible positions. Using components with only two states gives modern computers an internal simplicity that has enabled the development of larger, faster, and more reliable machines. Despite those advantages, the decision to adopt binary technology was not obvious to everyone in the early days in the field, as you will learn on the next page.
The binary vs. decimal debate

Although the use of the binary system is now standard in all modern computers, the idea took some time to catch on in the early years, particularly in the United States. Although early European computers (including the Manchester Baby described in Chapter 6 and several different machines designed by Konrad Zuse in Germany) used binary arithmetic, computing pioneers in the United States tended to favor decimal arithmetic, despite the fact that implementing computation in base-10 is substantially more complex.

One of the computer scientists who resisted the adoption of the binary system was Howard Aiken who, along with a team at Harvard University that included Grace Murray Hopper, built the Automatic Sequence Controlled Calculator, more commonly known as the Mark I. In a conversation with Henry Tropp, a historian with the Smithsonian in Washington, DC, Aiken defended his preference as follows:

I've always thought that building binary computers for commercial applications was a mistake. You spend so much time translating from the decimal to binary number system and back that you would have been better off to do it in the decimal system. I argued that building binary computers is a concession to the designer to simplify his job—and the designer’s job—is done in a few months, but the user’s job goes on forever.

Although the precise connections are uncertain, the design of the Mark I suggests that Aiken and Hopper were also influenced by Charles Babbage’s ideas from over a century earlier. Like Babbage’s machines, the Mark I stored numbers using a series of rotating wheels with ten positions, one for each decimal digit. The passage of time, of course, allowed some progress.

While Babbage dreamed that his calculating engines might someday be powered by steam; the calculating wheels of the Mark I were powered by electricity.

The fact that so many of the leading computing experts of his day shared Aiken’s attitude suggests that the transition from decimal to binary represents a “paradigm shift” of the sort Thomas Kuhn describes in The Structure of Scientific Revolutions. As the early British computer scientist Maurice Wilkes observed, “very few people of Aiken’s generation developed green fingers for electronics.” It was the task of a new generation—young people who had grown up with that new technology—to guide computing into the future.

The room-sized Aiken Automatic Sequence Controlled Calculator (Mark I)
4.1 Binary notation

The idea of writing numbers in binary notation predates the development of the electronic computer by more than 250 years. The German mathematician Gottfried Wilhelm von Leibniz (1646–1716) offered a detailed account of the binary system in a paper published by the French Royal Academy of Science in 1703. The introductory paragraphs and the first table from the paper appear in Figure 4-1.

**Figure 4-1** Excerpts from Leibniz's 1703 paper in *Memoires de l'Academie Royale des Sciences*
Even though the text is written using the typography of the early 18th century in which the letter s appears as /, the sense of the passage is remarkably easy to follow, even if you don’t speak French. Leibniz tell us

Ordinary arithmetic calculation is performed following a progression by tens. One uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, which signify zero, one, and the following numbers up to nine, inclusive. On going up to ten, one starts again, and writes ten as 10; ten times ten, or one hundred, as 100; ten times one hundred, or one thousand, as 1000; and ten times a thousand as 10000. And so on.

But instead of the progression by tens, I have used for several years the simplest progression of all, which goes by twos, which I find to be the perfection of the science of numbers. I therefore do not use any characters other than 0 and 1, and on going up to two, I start again. That is why two is written here as 10; and two times two or four as 100; and two times four or eight as 1000 . . .

As Leibniz’s description makes clear, it is easy to translate a number written in binary back to its decimal equivalent. All you need to do is add together the place values of each digit in the number, keeping in mind that each digit in a binary number counts for twice as much as its neighbor on the right. For example, if Leibniz were to use binary notation to represent the year of his birth, he would write the number down like this:

```
1 1 0 0 1 1 0 1 1 1 0
```

The following diagram shows that this value indeed corresponds to the value 1646:
For the most part, numeric representations in this book use decimal notation for readability. If the base is not clear from the context, the text follows the usual strategy of using a subscript to denote the base. For example, the equivalence of the binary value $11001101110_2$ and the decimal value 1646 can be made explicit by writing the numbers like this:

$$11001101110_2 = 1646_{10}$$

The key point in this example is that the number itself is always the same; the numeric base affects only the representation. Leibniz is careful in his paper to express numbers as words when there might otherwise be ambiguity. The number in this example is *one thousand six hundred and forty-six* no matter how you write it down. Numbers don’t have bases; representations do.

To give yourself practice using binary notation, convert the following values from the specified base to decimal or binary, as appropriate:

(a) $101010_2$
(b) $11111111_2$
(c) $10000101010_2$
(d) $17_{10}$
(e) $100_{10}$
(f) $1776_{10}$

### 4.2 Arithmetic using binary numbers

The examples of binary representation from the preceding section can easily make it seem that numbers expressed in binary are cumbersome. If nothing else, binary representations are certainly longer than their decimal equivalents, typically by a factor of three. They are, of course, also much less familiar. For human readers, the notation $11001101110_2$ is much harder to interpret than 1646. At first glance, it seems as if the early skeptics about binary notation may have had a point.

The real advantage of binary representation becomes evident when you consider the complexity of arithmetic operations. Sometime early in your school days, you were probably forced to learn the multiplication tables. Even if you were expected to master only the products from $0 \times 0$ up to $9 \times 9$, you still had to memorize a table with 100 entries. When you are working with binary numbers, the multiplication table has only four entries, one for each of the products $0 \times 0$, $0 \times 1$, $1 \times 0$, and $1 \times 1$. The tables for binary addition, subtraction, and multiplication appear in Figure 4-2 along with several examples of the steps involved in applying these operations to binary values, all of which come from Leibniz’s 1703 paper.

It turns out that simplifying the rules of arithmetic are just as valuable for computers as they are for humans. As you will discover later in this chapter, binary
arithmetic is straightforward to implement using electronic circuits in which each bit is modeled as a tiny switch that can be in either of two positions, one of which is designated as 0 and the other is designated as 1. It is entirely possible—and indeed the designers of the Mark I described at the beginning of the chapter did so—to build switching circuits in which the individual elements have ten positions rather than two. The problem is that implementing the rules for arithmetic becomes much harder when the components have more possible states. Designing circuits using binary logic makes it possible to create hardware that is simpler, smaller, faster, and easier to scale.

Using Leibniz’s examples as a model, complete the following calculations, showing both the binary calculations and their decimal equivalents:

(a) \[10001 + 11001\]  
(b) \[1100100 - 1001011\]  
(c) \[1011011 \times 10011\]

### 4.3 Assembling bits into larger units

Even though every value stored inside a computer is ultimately stored as bits, single bits hold so little information that they are rarely used on their own. To make it
easier to store information such as numbers or characters, individual bits are collected together into larger units that are then treated as integral units of storage. The smallest such combined unit is called a byte, which consists of eight bits. On most machines, bytes are then assembled into larger structures called words, where a word is usually defined to be the size required to hold an integer value of the type most appropriate for the hardware. Today, machines typically organize their memory into words that are either four or eight bytes long (32 or 64 bits). To make it easier to draw diagrams that fit within the margins of the page, this book assumes that words are four bytes long.

**Machine sizes**

The amount of memory available on a computer system varies over a wide range. Early machines supported memories whose size was measured in kilobytes (KB), the machines of the 1980s and ’90s had memory sizes measured in megabytes (MB), and today’s machines typically have memories measured in gigabytes (GB). In most sciences, the prefixes kilo, mega, and giga stand for one thousand, one million, and one billion, respectively. In the world of computers, however, those base-10 values do not fit well into the internal structure of the machine. By tradition, therefore, these prefixes are taken to represent the power of two closest to their traditional interpretations. Thus, in programming, the prefixes kilo, mega, and giga have the following meanings:

\[
\begin{align*}
\text{kilo (K)} & = 2^{10} = 1,024 \\
\text{mega (M)} & = 2^{20} = 1,048,576 \\
\text{giga (G)} & = 2^{30} = 1,073,741,824
\end{align*}
\]

A 64KB computer from the early 1970s would have had \(64 \times 1024\) or 65,536 bytes of memory. Similarly, a modern 4GB machine would have \(4 \times 1,037,741,824\) or 4,294,967,296 bytes of memory.

**Storing integers in bytes and words**

The binary representation described by Leibniz makes it easy to store nonnegative integers in bytes and words. An eight-bit byte is large enough to hold a integer value in the range 0 to 255, which is \(2^8 - 1\). Similarly, a 32-bit word used to represent an unsigned integer can hold values between 0 and 4,294,967,296, which is \(2^{32} - 1\).

Consider, for example, the eight-bit byte containing the following binary digits:

\[00101010\]
That sequence of bits represents the number forty-two, which you can verify—just as Leibniz would have done—by calculating the contribution for each of the individual bits, as follows:

$$\begin{align*}
0 & \times 1 = 0 \\
0 & \times 2 = 2 \\
1 & \times 4 = 0 \\
0 & \times 8 = 8 \\
1 & \times 16 = 0 \\
0 & \times 32 = 32 \\
0 & \times 64 = 0 \\
1 & \times 128 = 0 \\
\hline 
256 & = 42
\end{align*}$$

Because integers larger than 255 will not fit in a single byte, it is necessary to store such values in a larger unit. If, for example, you need to store the number of possible rotor settings from the Enigma machine described in Chapter 11, which is 1,054,560 (26 × 26 × 26 × 60), you could do so as follows:

$$\begin{align*}
\text{10100000} & \text{ 00010000} \text{ 00010111} \text{ 01100000} \\
\hline 
32 & \\
64 & \\
256 & \\
512 & \\
1024 & \\
4096 & \\
1048576 & \\
1054560 &
\end{align*}$$

**Representing negative numbers**

The examples in the preceding section show how positive integers can be expressed in binary notation but offer no sense as to how one might represent a negative integer. Early architectures often did so by reserving the most significant bit in each memory cell to represent the sign. In this model, which is called sign-magnitude notation, a positive value and its negative counterpart have the same internal representation except for the sign bit. For example, the eight-bit integer –42 looks like this in sign-magnitude notation:

$$10101010$$

In modern architectures, sign-magnitude notation has largely been superseded by twos-complement notation, in which a negative number is derived from its positive counterpart using a two-step process. The first step is to complement the bits in the original value, which consists of replacing each 0 with a 1 and each 1 with a 0. The
second step is to add one to the result. Thus, to form the twos-complement representation of $-42$, you begin by complementing each bit in the number 42. Given that the binary representation of 42 is

```
0 0 1 0 1 0 1 0
```

complementing each bit gives the following bit pattern:

```
1 1 0 1 0 1 0 1
```

The final step is to add one to this value, which produces the twos-complement form:

```
1 1 0 1 0 1 1 0
```

The primary advantage of twos-complement notation is that it requires no special circuitry to implement addition and subtraction inside the hardware of the machine. All you have to do is apply the standard mathematical operations and discard any carries that extend beyond the boundaries of the number. The following calculation, for example, shows that applying the standard addition process to 42 and $-42$ yields a result of 0, just as you would expect:

```
0 0 1 0 1 0 1 0
+ 1 1 0 1 0 1 1 0
```

(a) Determine the twos-complement representations of the integers $-1$, $-2$, and $-65$?

(b) Find the decimal value of the following number, which is expressed in twos-complement form:

```
1 0 0 0 0 0 0 1
```

**Hexadecimal notation**

Although this diagram makes it clear how to store a small integer value in a byte, it also helps to demonstrate the fact that writing numbers in binary form is terribly inconvenient. Binary numbers are cumbersome, mostly because they tend to be so long. Decimal representations are intuitive and familiar but make it harder to understand how the number translates into bits.
For applications in which it is useful to understand how a number translates into its binary representation without having to work with binary numbers that stretch all the way across the page, computer scientists tend to use hexadecimal (base 16) notation instead. In hexadecimal notation, there are sixteen digits that represent the values from 0 to 15. Although the decimal digits 0 through 9 are perfectly adequate for the first ten digits, classical arithmetic does not define the extra symbols you need to represent the remaining six. Computer science traditionally uses the letters A through F for this purpose, as follows:

\[
\begin{align*}
A &= 10 \\
B &= 11 \\
C &= 12 \\
D &= 13 \\
E &= 14 \\
F &= 15
\end{align*}
\]

What makes hexadecimal notation useful is the fact that you can easily convert between hexadecimal values and the underlying binary representation. All you need to do is combine the bits into groups of four. For example, the number forty-two can be converted from binary to hexadecimal like this:

\[
00101010
\]

The first four bits represent the number 2, and the next four represent the number 10. Converting each of these to the corresponding hexadecimal digit gives 2A as the hexadecimal form. You can then verify that this number still has the value 42 by adding up the digit values, as follows:

\[
2 \times 16 = 32 \\
A \times 1 = 10
\]

\[
2A = 42
\]

(a) Express the following decimal integers in hexadecimal form: 2, 255, and 65261.
(b) Determine the decimal equivalents of the hexadecimal values 64, 6C1, and BEAD.
(c) The compiled version of every Java program begins with the following 16 bits:

\[
1100101011111110
\]

How would you express that value in hexadecimal?
For the most part, numeric representations in this book use decimal notation for readability. If the base is not clear from the context, the text follows the convention of using a subscript to denote the base. Thus, the three most common representations for the number forty-two—decimal, binary, and hexadecimal—look like this:

\[ 42_{10} = 00101010_2 = 2A_{16} \]

The most important thing to remember, however, is that the number itself is always the same; the numeric base affects only the representation. Forty-two has an intrinsic meaning that is independent of the base, which is perhaps easiest to see in the representation an elementary school student might use:

The number of tick marks in this representation is forty-two. The fact that a number is written in binary, decimal, or any other base is a property of the representation, not of the number itself. Numbers do not have bases; representations do.

### 4.4 Representing nonnumeric data

So far, the discussion of binary notation has focused on the representation of integers. Although integers are essential to most forms of computation, they are by no means the only form of data or even the most interesting. Computers today work with many kinds of data, including text, images, sounds, and video. Inside the hardware, all of those data types are represented as a sequence of bits.

The notion that numeric data could be used to represent nonnumeric concepts is one of the foundational ideas of computer science. Doron Swade, who led the effort to rebuild Babbage’s Difference Engine for the Science Museum in London, credits this insight to Ada Lovelace. In an interview included in the film To Dream Tomorrow directed by John Füegi and Jo Francis, Swade describes Lovelace’s contribution as follows:

Ada saw something that, in some sense, Babbage failed to see. In Babbage’s world, his engines were bound by number. He saw that the machines could do algebra in the narrow sense that they could manipulate plus and minus signs. But his calculating engines—his Difference Engine and his Analytical Engine, which is the programmable, general-purpose machine—all were bound by number. They manipulated number as a manifestation of quantity. What Lovelace saw—what Ada Byron saw—was that number could represent entities other than quantity. So, once you had a machine for manipulating numbers, if those numbers represented other things—letters, musical notes—then the machine could manipulate symbols of which number was one instance.
The principle of enumeration

The simplest strategy for representing nonnumeric data is to assign distinct integers to the individual data values you need to represent. For example, the conventional way to represent the months of the year—even without a computer—is to give each month a number: January has the value 1, February has the value 2, and so on, up to December, which has the value 12. This strategy is called enumeration.

Once you have enumerated a set of values, you can represent those values in memory by using the appropriate numeric code. For example, the numeric value 12 that corresponds to the month of December can easily be represented in a byte as follows:

```
0 0 0 0 1 1 0 0
```

The most important thing to recognize about this mode of representation is that there is no indication in the hardware as to whether the value of this byte is the integer 12 or the numeric representation for the month of December. The meaning of a particular set of bits depends on how that value is used. If the program uses the value arithmetically, that byte has the value 12. If it instead uses that value to select from a list of month names, that byte indicates December. In either case, the bit pattern is exactly the same.

Representing characters

Although the early computers developed in the mid-20th century worked primarily with numbers, modern computers manipulate many other kinds of data, all of which are represented internally as bits. Of these nonnumeric data types, one of the most important is text data, which is composed of the characters that appear on the keyboard and the screen. The ability of modern computers to process text data has led to the development of word processing systems, online reference libraries, electronic mail, social networks, and a seemingly infinite supply of applications.

The most primitive elements of text data are individual characters. Like the months of the year, characters can be represented inside the computer by assigning each character a numeric code. You could, for example, assign successive integers to represent each of the letters in the alphabet, using 0 for the letter A, 1 for letter B, and so on. In 1605, the English philosopher and scientist Francis Bacon did precisely that when he devised a technique for encoding messages that is now known as Bacon's cipher. What is, however, even more astonishing is that Bacon based his cipher on the binary representation of these numbers, almost a century before Leibniz published his paper on binary arithmetic. Bacon's cipher, however, was not used in practice and had little or no influence on the later development of computation.
The first binary encoding scheme for characters used extensively in practice was the Baudot code, which was invented in 1870 by the French engineer Émile Baudot, one of the pioneers of the telegraph. In Baudot’s scheme, each of the 26 letters was assigned a five-bit binary code, as shown in the table on the right. The encoding also included a few special characters to represent spaces, the two characters telegraph printers used to designate the end of a line, and transitions to an alternate character set used for digits and punctuation. The letters of the alphabet did not appear in order, but were instead chosen so that the most common letters, such as E and T, would require pressing just one of the five keys on the input device.

The fact that the letters do not appear consecutively in the Baudot code does not make it any less effective as an encoding scheme. The only essential characteristic of an encoding scheme is that the sender and receiver agree on how to convert letters to numeric codes. The need for a common encoding shared by senders and receivers increases the importance of standardization. As long as all telegraph operators used the same code, they were able to communicate with one another.

In the early years of the computing industry, standardization was complicated by the existence of two incompatible character encodings. The American Standards Association (now known as the American National Standards Institute or ANSI) began work on a standardized character encoding in 1960, which was formalized in 1963 as the American Standard Code for Information Interchange or ASCII. Early IBM machines, however, used a different character set derived from the coding system used for devices that worked with punched cards. When it released the System/360 operating system in 1964, IBM—at that time the undisputed industry leader—chose to create a competing coding scheme called the Extended Binary Coded Decimal Interchange Code or EBCDIC, which was more compatible with existing IBM devices. The two encoding schemes coexisted for several decades, but ASCII and its successors have become nearly universal in recent years.

In its original design, ASCII was a seven-bit code, which allows for 128 (2^7) characters, which is enough to store the uppercase and lowercase letters of the Roman alphabet, the standard decimal digits, a variety of punctuation symbols, and a set of nonprinting characters called control characters, most of which have lost their original meaning. The characters in the ASCII set appear in Figure 4-3. The row and column headers show the hexadecimal digits making up the character code. The ASCII code for the uppercase A, for example, is the hexadecimal value 41, and the code for the question mark character is the hexadecimal value 3F.

The ASCII coding system, however, was developed in the United States and was in many ways specific to that environment. For example, the character set includes the currency symbol for the U.S. dollar, but omits symbols like £, €, and ¥ used for
other major currencies in the world. It is, moreover, closely tied to the Roman alphabet. Particularly with the rise of the World Wide Web in the 1990s, it was necessary to expand the encoding system to embrace a broader collection of languages. The result of that expansion was a new standard called Unicode, which is designed to be much more universal in its application. The original versions of Unicode allowed for 65,536 (2^16) characters, but even that number proved to be insufficient. The current Unicode standard allows for 1,114,112 characters, which allows it to represent not only the languages that are in use today, but also the languages of antiquity, including, for example, the early Mycenaean syllabic script called Linear B, which was the forerunner of the Greek alphabet, as shown in Figure 4-4. There are even proposals to include in the Unicode standard scripts from fantasy, such as J.R.R. Tolkien’s Tengwar script from Lord of the Rings.
4.5 Strings in JavaScript

The JavaScript programs you saw in Chapter 2 worked only with numeric data. Programs, however, become more interesting if they can manipulate a wider range of information. Now that you’ve seen how text data can be represented as collections of bits, it is useful to learn how to use text data in your programs.

In most applications, characters do not appear individually but are instead combined to form a string, which is simply a sequence of characters. Strings are a predefined type in JavaScript, and you can include strings in a JavaScript program simply by enclosing the characters in quotation marks. For example, "hello, world" is a string composed of 12 characters including ten letters, a comma, and a space. The quotation marks are not part of the string but instead serve to indicate where it begins and ends.

In certain respects, JavaScript allows you to use string values in much the same ways that you use numeric values. You can, for example, store strings in a variable or pass them as arguments to a function. For example, the declaration

```
var str = "hello, world";
```

sets the variable `str` to the 12-character string "hello, world". Similarly, the declaration

```
var ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

sets the constant `ALPHABET` to a 26-character string containing the uppercase letters.

The difference between strings and numbers lies in the operations you can perform on values of these types. As in most modern languages, JavaScript offers a rich set of operations that extends far beyond the level that makes sense for this book. Even so, it is important for you to understand a few string operations so that you can use them to create more interesting programs.

**String operators**

In Chapter 2, you learned that you can use the + operator to join strings together end to end. In programming contexts, this operation is called concatenation. If you apply the + operator to two numbers, JavaScript adds them numerically. If the value on either side of the + operator is a string, JavaScript converts both values into strings and concatenates the two. Thus, the expression `2 + 2` has the numeric value 4, while the expression "Catch " + 2 + 2 has the string value "Catch22".
In addition to the `+` operator, JavaScript allows you to use the relational operators `===`, `!==`, `<`, `<=`, `>`, and `>=` to compare two string values. For example, you can use the following code to check whether the value of the variable `str` is equal to "quit":

```javascript
if (str === "quit") . . .
```

The relational operators compare strings using **lexicographic order**, which is similar to traditional alphabetical order but which uses the underlying Unicode values of each character to make the comparison. Lexicographic order means that case is significant, so "a" is not equal to "A". In lexicographic order, "a" is greater than "A" because the Unicode value for a lowercase `a` (hexadecimal 61) is greater than the Unicode value for an uppercase `A` (hexadecimal 41).

### Strings and the object-oriented paradigm

If you need to perform more complex operations on strings, you will need to learn some new aspects of JavaScript syntax. For the most part, JavaScript string operations are designed to match those provided by Java so that programmers don’t have to learn two radically different models when working in these languages. Java adopts a philosophy of programming called the **object-oriented paradigm**, which focuses on encapsulating data and operations together into a unified structure. In keeping with the object-oriented paradigm, Java—and by extension JavaScript, which adopted the same design—implements strings as a **class**, which is most easily defined informally as a template for a set of values and an associated set of operations. The values that belong to a class are called **objects**. A single class can give rise to many different objects; each such object is said to be an **instance** of that class.

In programming languages that adopt the object-oriented paradigm, objects communicate by sending information and requests from one object to another. Collectively, these transmissions are called **messages**. The act of sending a message corresponds to having one object invoke a function defined by another object. In the terminology of object-oriented programming, functions that belong to an object are called **methods**. For consistency with the conceptual model of sending messages, the object that initiates the method is called the **sender**, and the object that is the target of that transmission is called the **receiver**.

In both Java and JavaScript, sending a message to an object is specified using the syntax

```javascript
receiver . name (arguments)
```

where `receiver` is the object receiving the message, `name` is the name of the method that implements that message, and `arguments` is the list of values used to initialize
the parameters of the method, just as in any function call. JavaScript uses the object-oriented model to implement string operations, which means that you need to use the receiver syntax to invoke these operations.

Figure 4-5 lists several of the most common methods that JavaScript defines as part of its string class. These methods are explored in more detail in the individual sections that follow.

**Determining the length of a string**

The simplest operation that you can perform on a string value is determining its length, which is the number of characters it contains. Given a JavaScript string variable `str`, you can determine the length by evaluating `str.length`. It is important to note that, in contrast to the other string operations listed in Figure 4-5, `length` is not defined as a method but instead as a JavaScript property, which is a

---

### Figure 4-5 Common operations in the String class

<table>
<thead>
<tr>
<th>String operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str1 + str2</code></td>
<td>Concatenates <code>str1</code> and <code>str2</code> end to end and returns a new string containing the combined characters. As long as one operand is a string, Java will convert the other operand to its string form.</td>
</tr>
<tr>
<td><code>str += suffix</code></td>
<td>Appends <code>suffix</code> to the end of <code>str</code>.</td>
</tr>
<tr>
<td><code>str1 === str2</code></td>
<td>These operators compare <code>str1</code> and <code>str2</code>. The comparison is performed using lexicographic order, which is the order defined by the underlying character codes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str.length</code></td>
<td>Indicates the number of characters in <code>str</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str.charAt(k)</code></td>
<td>Returns a one-character string containing the character at index position <code>k</code> in <code>str</code>.</td>
</tr>
<tr>
<td><code>str.substring(p1, p2)</code></td>
<td>Returns a new string of characters beginning at <code>p1</code> in <code>str</code> and extending up to but not including <code>p2</code>. If <code>p2</code> is missing, the new string continues through the end of the original string.</td>
</tr>
<tr>
<td><code>str.indexOf(pattern)</code></td>
<td>Searches the string <code>str</code> for <code>pattern</code>. The search starts at the beginning, or at index <code>k</code>, if specified. The function returns the first index at which <code>pattern</code> appears, or −1 if it is not found.</td>
</tr>
<tr>
<td><code>str.lastIndexOf(pattern)</code></td>
<td>Operates like <code>indexOf</code>, but searches backward from position <code>k</code>. If <code>k</code> is missing, <code>lastIndexOf</code> starts at the end of the string.</td>
</tr>
<tr>
<td><code>str.toLowerCase()</code></td>
<td>Returns a copy of <code>str</code> converting all characters to lowercase.</td>
</tr>
<tr>
<td><code>str.toUpperCase()</code></td>
<td>Returns a copy of <code>str</code> converting all characters to uppercase.</td>
</tr>
</tbody>
</table>
data value associated with an object. Defining `length` as a property means that no parentheses appear after the property name.

As an example, if `str` and `ALPHABET` have been initialized as shown on page 85, the expression `str.length` has the value 12, and `ALPHABET.length` has the value 26.

**Selecting characters from a string**

In JavaScript, positions within a string are numbered starting from 0. For example, the characters in `ALPHABET` are numbered as in the following diagram:

```
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19| 20| 21| 22| 23| 24| 25|
```

The position number written underneath each character is called its **index**.

In JavaScript, you select a character from a string by calling the `charAt` method. For example, the expression

```
ALPHABET.charAt(0)
```

selects the character "A" at the beginning, which is returned as a one-character string. Since character numbering in JavaScript begins at 0, the last character in a string appears at the index position that’s one less than the length of the string. Thus, you can select the "Z" at the end of `ALPHABET` with the following expression:

```
ALPHABET.charAt(ALPHABET.length - 1)
```

**Extracting parts of a string**

While concatenation makes longer strings from shorter pieces, you often need to do the reverse: separate a string into the shorter pieces it contains. A string that is part of a longer string is called a substring. JavaScript’s string class exports a method called `substring` that takes two parameters: the index of the first character you want to select and the index of the character that immediately follows the desired substring. For example, the method call

```
ALPHABET.substring(1, 4)
```

returns the three-character substring "BCD". Because indices in JavaScript begin at 0, the character at index position 1 is the character "B".

The second argument in the `substring` method is optional. If it is missing, the `substring` method returns the substring that starts at the specified position and continues through the end of the string. Thus, calling
returns the string "XYZ".

Java’s string library includes two methods, `startsWith` and `endsWith`, that are surprisingly useful in writing string applications. Although these methods have been included in the most recent JavaScript standard, they remain unimplemented in many browsers. Fortunately, it is easy to correct this deficiency by writing your own functions to accomplish this effect.

Implement predicate functions `startsWith(str, prefix)` and `endsWith(str, suffix)` that return `true` if the value of `str` starts with the specified prefix or ends with the specified suffix, as appropriate. For example, calling `startsWith("catastrophe", "cat")` should return `true`.

**Searching within a string**

From time to time, you will find it useful to search a string to see whether it contains a particular character or substring. To support such search operations, JavaScript’s string class exports a method called `indexOf`, which comes in several forms. The simplest form of the call is

```
str.indexOf(pattern);
```

where `pattern` is the content you’re looking for. When called, the `indexOf` method searches through `str` looking for the first occurrence of the pattern. If the search value is found, `indexOf` returns the index position at which the match begins. If the character does not appear before the end of the string, `indexOf` returns −1.

The `indexOf` method takes an optional second argument that indicates the index position at which to start the search. The effect of both styles of the `indexOf` method is illustrated by the following examples, which assume that the variable `str` contains the string "hello, world":

```
str.indexOf("o") → 4
str.indexOf("o", 5) → 8
str.indexOf("o", 9) → -1
```

The JavaScript string class also includes a `lastIndexOf` method that works like `indexOf`, except that it searches backward for a match, starting from the specified index position or from the end of the string in the single-argument case.
**Case conversion**

The methods `toLowerCase` and `toUpperCase` convert any alphabetic characters in the receiver string to the specified case, leaving any other characters unchanged. For example, if `str` contains "hello, world", calling `str.toUpperCase()` returns "HELLO, WORLD". Similarly, calling `ALPHABET.toLowerCase()` returns "abcdefghijklmnopqrstuvwxyz".

It is important to remember that the methods in JavaScript’s string class do not change the value of the receiver but instead return an entirely new string value. Thus, calling `str.toUpperCase()` doesn’t change the value of the variable `str`. If you want to change the value of `str` to its uppercase equivalent, you need to use an assignment statement to store the value back into the variable, as in

```javascript
str = str.toUpperCase();
```

The `toUpperCase` method makes it easy to write a predicate function called `equalsIgnoreCase` that checks whether two strings are equal if the comparison ignores the distinction between uppercase and lowercase characters, as follows:

```javascript
function equalsIgnoreCase(s1, s2) {
    return s1.toUpperCase() === s2.toUpperCase();
}
```

You can also use the `toUpperCase` method together with `indexOf` and the `ALPHABET` constant from page 85 to implement a predicate function `isLetter` that checks to see whether its argument is a single letter, as follows:

```javascript
function isLetter(ch) {
    return ch.length === 1 &&
        ALPHABET.indexOf(ch.toUpperCase()) !== -1;
}
```

The second part of the test checks to see if the uppercase version of the parameter `ch` appears anywhere in the `ALPHABET` string.

Implement a function `capitalize(str)` that returns a string in which the initial character is capitalized and all other letters are converted to lowercase. Characters other than letters should remain unchanged. For example, both `capitalize("BOOLEAN")` and `capitalize("boolean")` should return the string "Boolean".
4.6 Common string patterns

Even though the methods exported by JavaScript’s string class provide the tools you need to implement simple string applications, it is usually easier to write programs by adapting code patterns that implement particularly common operations. The two most important patterns are iterating through the characters in a string and growing a string by concatenation. These patterns are outlined in more detail in the sections that follow.

As with other important programming patterns, it is probably best to memorize these string patterns so that you don’t have to reinvent them. Having the patterns in your head means that you can focus your attention on the problem at hand.

Iterating through the characters in a string

When you work with strings, one of the most important patterns involves iterating through the characters in a string, which requires the following code:

```javascript
for (var i = 0; i < str.length; i++) {
  . . . body of loop that uses the character str.charAt(i) . . .
}
```

On each loop cycle, the expression `str.charAt(i)` refers to the \( i \)th character in the string. Because the purpose of the loop is to process every character, the loop continues as long as \( i \) is less than the length of the string. Thus, you can count the number of spaces in a string using the following function:

```javascript
function countSpaces(str) {
  var nSpaces = 0;
  for (var i = 0; i < str.length; i++) {
    if (str.charAt(i) === " ") nSpaces++;
  }
  return nSpaces;
}
```

For some applications, you will find it useful to iterate through a string in the opposite direction, starting with the last character and continuing backward until you reach the first. This style of iteration uses the following for loop:

```javascript
for (var i = str.length - 1; i >= 0; i--)
```

Here, the index \( i \) begins at the last index position, which is one less than the length of the string, and then decreases by one on each cycle, down to and including the index position 0.
Assuming that you understand the syntax and semantics of the for statement, you could work out the patterns for each iteration direction from first principles each time this pattern comes up in an application. Doing so, however, would slow you down enormously. These iteration patterns are worth memorizing so that you don’t have to waste any time thinking about them. Whenever you recognize that you need to cycle through the characters in a string, some part of your nervous system between your brain and your fingers should be able to translate that idea effortlessly into the following line:

```javascript
for (var i = 0; i < str.length; i++)
```

### Growing a string through concatenation

The other string pattern that it’s important to memorize involves creating a new string one character at a time. The loop structure itself will depend on the application, but the general pattern for creating a string by concatenation looks like this:

```javascript
var str = "";
for (whatever loop header line fits the application) {
  str += the next substring or character;
}
```

As a simple example, the following method returns a string consisting of \( n \) copies of the string \( str \):

```javascript
function nCopies(n, str) {
  var result = "";
  for (var i = 0; i < n; i++) {
    result += str;
  }
  return result;
}
```

The \( n \)Copies function is useful if, for example, you need to generate some kind of section separator in console output. One strategy to accomplish this goal would be to use the statement

```javascript
Console.println(nCopies(72, "-"));
```

which prints a line of 72 hyphens.
Combining the iteration and concatenation patterns

Many string-processing methods use the iteration and concatenation patterns together. For example, the following method reverses the argument string so that, for example, calling `reverse("stressed")` returns "desserts":

```javascript
function reverse(str) {
    var result = "";
    for (var i = str.length - 1; i >= 0; i--) {
        result += str.charAt(i);
    }
    return result;
}
```

You could also implement `reverse` by running the loop in the forward direction and concatenating each new character to the front of the `result` string, as follows:

```javascript
function reverse(str) {
    var result = "";
    for (var i = 0; i < str.length; i++) {
        result = str.charAt(i) + result;
    }
    return result;
}
```

An **acronym** is a word formed by combining, in order, the initial letters of a series of words. For example, the word *scuba* is an acronym formed from the first letters in *self-contained underwater breathing apparatus*. Similarly, *AIDS* is an acronym for *Acquired Immune Deficiency Syndrome*. Write a method `acronym` that takes a string and returns the acronym formed from that string. To ensure that your method treats hyphenated compounds like *self-contained* as two words, it should define the beginning of a word as any alphabetic character that appears either at the beginning of the string or after any character other than a letter, which you can determine by calling the `isLetter` function defined on page 90.

Recognizing palindromes

A **palindrome** is a word that reads identically backward and forward, such as *level* or *noon*. The goal of this section is to write a predicate function `isPalindrome` that checks whether a string is a palindrome. Calling `isPalindrome("level")` should return `true`; calling `isPalindrome("xyz")` should return `false`. 
As with most programming problems, there is more than one strategy for solving this problem. In my experience, the approach that most students are likely to try first uses a `for` loop to run through each index position in the first half of the string. At each position, the code then checks to see whether that character matches the one that appears in the symmetric position relative to the end of the string. Adopting that strategy leads to the following code:

```javascript
function isPalindrome(str) {
    var n = str.length;
    for (var i = 0; i < n / 2; i++) {
        if (str.charAt(i) != str.charAt(n - i - 1)) {
            return false;
        }
    }
    return true;
}
```

If you make use of the functions you already have, you can code `isPalindrome` in a much simpler form, as follows:

```javascript
function isPalindrome(str) {
    return str === reverse(str);
}
```

Of these implementations, the first is substantially more efficient. The second implementation constructs the reversed string by concatenation, which requires the creation of several new strings. The first version works by selecting and comparing characters, which are less costly operations.

Despite this difference in efficiency, the second version has many advantages, particularly as an example for new programmers. For one thing, it takes advantage of existing code by making use of the `reverse` function. For another, it hides the complexity involved in calculating index positions required by the first version. It takes at least a minute or two for most students to figure out why the code includes the selection expression `str.charAt(n - i - 1)` or why it is appropriate to use the `<` operator in the `for` loop test, as opposed to `<=`. By contrast, the line

```
return str === reverse(str);
```

reads as fluidly as English: a string is a palindrome if it is equal to the same string if you reverse it. That, after all, is precisely the definition of a palindrome.

Particularly as you are learning about programming, it is much more important to work toward the clarity of the second implementation than the efficiency of the first. Given the speed of modern computers, it is almost always worth sacrificing some efficiency to make a program easier to understand.
From your reading of Chapter 4, you have some idea of how collections of bits can be used to represent many different types of data. Even so, you are no closer to understanding how those bits are represented inside a machine. The purpose of this chapter is to explore how simple circuit elements can store and manipulate binary information.

Particularly if you often find yourself in awe of the seemingly magical power of modern computers, my use of the word simple in the preceding sentence may leave you feeling a bit skeptical. Computers are, after all, among the most complex artifacts that human beings have created. Even the smallest computers have vastly more possible states than there are atoms in the universe, and it is difficult to regard as simple any machine with that much inherent complexity. My point is that the elements that exist inside a computer are simple and that the complexity arises primarily from the scale at which those elements are replicated. Given today’s technology, a large microprocessor chip contains billions of transistors, each of which performs a simple function. When those simple functions are replicated in the massive numbers one sees of modern hardware, the result is an extraordinarily sophisticated machine capable of solving highly complex problems.

The relationship between the individual circuit elements and the computer as a whole is in some ways analogous to the relationship between atoms and a complex organism such as a human being. Everything in our bodies is composed of atoms chosen from a relatively small set of elements. The complexity of a human being comes from the scale at which those atoms are replicated. Each cell in the human body contains trillions of atoms, and the body itself contains trillions of cells.
The groundbreaking contributions of Claude Shannon

In the early years of computing, Howard Aiken’s opposition to using binary arithmetic in commercial machines was the dominant view. The most influential challenge to that orthodoxy came from a 21-year-old graduate student at the Massachusetts Institute of Technology whose 1937 thesis entitled “A Symbolic Analysis of Relay and Switching Circuits,” has often been described as the most influential Master’s thesis in history. In his paper, which was republished the following year in the prestigious Transactions of the American Institute of Electrical Engineers, Shannon showed how the operations of binary arithmetic could be implemented using commercially available circuit elements. This discovery—along with the theoretical arguments that Shannon used to establish its correctness—eventually led to the adoption of binary arithmetic in all modern architectures.

Shannon went on to make several other important contributions to computer science, primarily during the years that he worked as a researcher at Bell Labs in New Jersey, which was at the time one of the leading research centers in the world. In 1948, Shannon published a paper in which—in addition to introducing the word *bit* into the English language—he outlined a new mathematical theory of communication. That paper became the foundation of a field that we now call *information theory*, which looks at information as an abstract concept, independent of any meaning that we might assign to it. Information theory transformed the technology of communication and laid the groundwork for today’s networked world.

Shannon went on to make several other important contributions to computer science, primarily during the years that he worked as a researcher at Bell Labs in New Jersey, which was at the time one of the leading research centers in the world. In 1948, Shannon published a paper in which—in addition to introducing the word *bit* into the English language—he outlined a new mathematical theory of communication. That paper became the foundation of a field that we now call *information theory*, which looks at information as an abstract concept, independent of any meaning that we might assign to it. Information theory transformed the technology of communication and laid the groundwork for today’s networked world.

Shannon’s paper on information theory grew out of classified work in code-breaking that he had done during World War II. After the war, he continued to work in cryptography and was the first to prove mathematically that codes could be unbreakable.

Shannon made essential contributions in a variety of areas, many of which lie outside of computer science. His doctoral thesis, for example, applied his earlier work with binary logic to genetics. He made several early forays into the field of artificial intelligence, including writing an article for *Scientific American* on computerized chess and building an electromechanical mouse that could discover and then remember a path through a maze.

Throughout his life, Shannon never lost his sense of fun. He designed juggling machines and invented new types of unicycles, including one without a seat that he dubbed “The Ultimate Wheel.”
Unfortunately, understanding the structure and behavior of individual atoms offers little insight into the workings of the human body. Similarly, knowing how a transistor operates is by no means sufficient to explain the operation of a computer as a whole. In each case, it is necessary to view the processes and structures at multiple levels of detail. Atoms, for example, form molecules, which range in complexity from simple compounds like water up to the intricate structure of the DNA helix, which carries the blueprint for every human being. The DNA molecule controls the synthesis of protein molecules, which are the building blocks for cells. The cells then combine to form organs that in turn comprise the human organism. Biologists seek to understand an organism by studying its many levels independently.

In much the same way, understanding the hardware structure of a computer requires taking a hierarchical view. As you will see in this chapter, individual transistors—each of which behaves as a tiny on-off switch—are combined to form gates that perform basic logical functions. Those gates can be used to build simple circuits that, for example, store a single bit in memory or add two binary digits. Multiple copies of these simple circuits can be combined into a more complex circuit that stores an entire memory word or adds two binary numbers. These more complex circuits can then be replicated millions of times to create the inner workings of the computer as a whole.

The complexity of the overall hierarchy becomes easier to manage if each level is as simple as possible. Nature, for example, has somehow contrived that the DNA molecule, despite its enormous size, consists of only four bases—adenine, guanine, cytosine, and thymine—arranged in different sequences. Similarly, modern hardware typically contains millions or billions of gates, all of which are chosen from just a few possible types. The complexity comes, as it does in the DNA molecule, from how those individual elements are arranged and connected.

### 5.1 Switching circuits

The best place to start an exploration of hardware—as Claude Shannon did in his 1937 thesis—is with the behavior of switches just like those that turn the lights in a room on or off. In circuit diagrams, switches are represented using the following symbol, which shows the switch in its open or off position:

![Switch symbol](open-off)

Flipping the switch rotates the heavy bar so that it touches both contacts, as shown in the following diagram of the switch in its closed or on position:

![Switch symbol](closed-on)
When a switch is open, there is no connection between the two switch contacts, which means that no electrical current can flow across the switch. As a result, the lamp is off in the following circuit, in which the ⊗ the ⊘ symbols indicate the terminals of a battery:

Closing the switch allows current to flow, turning the lamp on:

**Series and parallel circuits**

Connecting multiple switches makes it possible to implement more sophisticated behavior. The following diagram shows two switches connected end-to-end, which electrical engineers call a *series circuit*:

In this circuit, the lamp is on only if both switches are closed. If either or both switches are in the open position, the lamp stays off. Thus, connecting two switches in series implements a logical operation analogous to the word *and* in English.

You can also connect two switches in the following arrangement, which electrical engineers call a *parallel circuit*:

In this circuit, the lamp is lit if either or both switches are closed. The parallel circuit therefore implements a logical operation analogous to the English word *or*.

In addition to the *and* and *or* operations represented by the series and parallel circuits, computer hardware sometimes needs to be able to detect when a switch is in the open position. The simplest way to achieve this goal is to add another output contact to the switch, which is connected when the switch is open, as follows:
If you connect the newly added contact in this switch with the lamp and battery terminals, you get the following circuit, in which current flows even though the switch is in the off position:

![Circuit Diagram](image)

Flipping the switch to the on position in this circuit breaks the connection, turning the lamp off, as follows:

![Circuit Diagram](image)

This switching arrangement implements the logical function analogous to the English word *not*. The circuit is on when the switch is off, and vice versa.

**Controlling a room light with two switches**

At some point in your life, you have almost certainly been in a room in which two switches, typically at opposite sides of the room, both control the ceiling light. The problem is represented schematically in the puzzle box at the bottom of this page. Your challenge is to add wires so that either switch controls the lamp no matter how the other switch is set. One solution appears on the next page, but it is worth spending a little time trying to solve the problem before you look at the solution.

![Puzzle Box](image)

Add wires to the circuit diagram so that the ceiling light is controlled by either of the two switches.
As a starting point, it is worth noting that neither of the circuits you have seen so far has the desired effect. Connecting the switches in series gets you partway to the goal but doesn’t completely solve the problem. If one switch is on in a series circuit, flipping the other one controls the light, just as you would hope. If either switch is off, however, flipping the other switch has no effect. The parallel circuit fails in the opposite way. If either switch is on, there is nothing you can do with the other switch to turn the light off.

What you need is a circuit that turns the light on if the two switches are in the same position or, equivalently, if they are in different positions. Each of these strategies implements the desired behavior. If two switches are in the same position, flipping either one leaves them in different positions. If they are in different positions, flipping either switch ensures that the switches are in the same position. The solution that turns the light on when the switches are in different positions appears in Figure 5-1.

The fact that the wires in Figure 5-1 run all the way around the room makes the operation of the circuit a little difficult to see. The circuit in this figure is equivalent to the following circuit in which the right-hand switch has been reversed to simplify the wiring:
In this circuit, the lamp is on whenever the switches are in different positions, as illustrated by the four diagrams in the right margin that show every possible combination of the switches. This circuit implements a logical operation that computer scientists call *exclusive or*, which is often shortened to *xor*. The name comes from the fact that the connection is complete if either the left switch or the right switch is in the *on* position, but not if both switches are.

The problem of controlling the ceiling light can also be solved using an *equivalence circuit*, which allows current to flow only when the two switches are in the same position, as follows:

Using room lights to explain parity circuits has a distinguished heritage. As you can see from Figure 5-2, Claude Shannon used exactly this analogy in his Master’s thesis, which has revolutionized the design of all computers ever since. The text in the figure also makes it clear that the exclusive-or and equivalence circuits can be generalized to solve the problem of controlling a room light using any number of

**Figure 5-2** Excerpt from Claude Shannon’s Master’s thesis illustrating parity circuits

These circuits each require 4(n-1) elements. They will be recognized as the familiar circuit for controlling a light from n points: If at any one of the points the position of the switch is changed, the total number of variables which equals zero is changed by one, so that if the light is on, it will be turned off and if already off, it will be turned on.
switches. The basic strategy is to use the switches to determine whether the total number of switches in the on position is even or odd. That number is called the *parity* of the circuit. A circuit like exclusive or is an *odd-parity circuit* because it is on when the number of switches in the on position is odd. By contrast, the equivalence circuit is an *even-parity circuit* because the circuit is on when the number of switches in the on position is even.

**Relays**

As Claude Shannon made clear in his Master’s thesis, simple switches are not sufficient in themselves to implement the fundamental operations required for computation. It is also essential to allow the output of one switching circuit to control the state of other parts of that circuit. The strategy that Shannon employed in 1937, which found its way into the title of his thesis, was to make use of *relays*, which are switches controlled by an electromagnet.

A simple version of a relay looks like this in schematic form:

![Relay schematic](image)

In this position, the switch arm above the relay coil connects to the top contact on the right-hand side. Passing current through the coil activates the electromagnet, which pulls the switch arm down so that it touches the bottom contact, as follows:

![Relay operation](image)

When current is removed from the coil, the switch flips back to its original position.

The following circuit illustrates how a relay can invert the behavior of a switch, so that the circuit is on when the switch is off, and vice versa:
When the switch is open, current flows through the switch bar of the relay and lights the lamp. Closing the switch sends current through the coil, which pulls the relay switch downward, turning the lamp off, as follows:

Although relays and switching circuits provide all the power you need to implement computation, they are far too slow to be used in modern hardware. Even so, the relay provides a useful conceptual model for the circuitry that is in fact used today. All modern hardware depends on the idea of an electronic switch, in which current flowing in one part of a circuit can control current in another. The underlying concepts are the same, no matter whether you implement digital logic using relays, vacuum tubes, transistors, or any functionally equivalent technology.

### 5.2 Logic gates

As with many different aspects of computer science, one of the most effective strategies for managing the complexity of hardware circuits is to move from the details of a particular implementation toward a more abstract perspective based on the overall behavior of that circuit. Ignoring low-level details to focus on the high-level properties of a system is an essential technique in all of computer science.

In circuit design, the easiest way to take this more abstract view is to use **logic gates**, which are abstract components that perform specific logical operations. Each gate has a certain number of input and output connections. Each of those connections carries a **signal**, which is a single binary value. Signals in a circuit are usually assigned names consisting of one or more uppercase letters. If a gate has two input signals, the usual convention is to label those signals A and B.

In the circuitry used for modern hardware, signals are represented as voltages, but the details of that representation are unimportant for the moment. For now, all you need to know is that every signal in a circuit carries one of two possible voltages, which are labeled using the binary digits 0 and 1.

**The AND, OR, and NOT gates**

Each of the switching circuits introduced in section 5.1 has a direct counterpart in the world of logic gates, which are identified by symbols of a particular shape. For
example, the series circuit used to model the English word *and* corresponds to the **AND** gate, which looks like this:

```
   0   0
   0   0
   1   0
   1   1
```

The output of the **AND** gate is 1 only if both inputs have the value 1. If either input signal is a 0 or if both of them are, the output of the **AND** gate is 0.

The parallel switching circuit that models the English word *or* corresponds to the **OR** gate, which is represented by the following symbol:

```
   0   0
   0   0
   0   1
   1   1
```

The output of the **OR** gate is 1 if either or both inputs have the value 1. If both input signals are 0, the output of the **OR** gate is 0.

The logical operation of inverting a signal is the function of the **NOT** gate, which is represented like this:

```
   0   1
   1   0
```

The output of this gate, which is also called an *inverter*, is 0 if the input is 1, and 1 if the input is 0.

The sidebar at the right shows the output for each of these standard gates with all possible input signals. The effect of these gates can be captured more concisely in an **operation matrix**, which shows the value of the output in a two-dimensional grid. For gates with two inputs, the binary value of the input A appears along the left side of the matrix and the binary value of the input B appears across the top. For gates with just one input like the **NOT** gate, the operation matrix has just one column that specifies the result. The operation matrices for the **AND**, **OR**, and **NOT** gates look like this:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Combining logic gates

One of the advantages of using logic gates is that it is easy to combine them to form more complex structures. For example, you can string together two AND gates to create a circuit that applies the and operation to three input signals, as follows:

Similarly, you can use two AND gates and two OR gates to produce a circuit that determines whether at least two of three input signals have the value 1. This combination of gates is called a majority circuit and looks like this:

Verify that the three-input AND circuit and the majority circuit produce the desired output by tracing all possible input signals through each network of gates.

Implementing binary addition

Digital logic supports other types of gates beyond the AND, OR, and NOT gates you have seen so far. One of the most useful is the XOR gate, which implements the exclusive-or function introduced in the earlier section on “Controlling a room light with two switches.” In circuit diagrams, the XOR gate looks like this:

As illustrated in the sidebar to the left, the output of the XOR gate is 1 if exactly one of the inputs is a 1. If both input values are 0 or both input values are 1, the output value of the XOR gate is 0. This behavior is summarized in the following matrix:
As it happens, you have already seen this matrix in Chapter 4. It shows up in Figure 4-2, which describes Leibniz’s process for binary arithmetic. The addition table for binary digits has exactly the same values, which means that you can use an XOR gate to compute the sum of two binary digits. Similarly, you can use an AND gate to determine whether there is a carry to the next digit position. Putting these two gates together creates a circuit called a half adder, which calculates the sum and carry values for two binary digits. The circuit for a half adder therefore has the following form:

![Half Adder Circuit Diagram]

In this circuit, A and B are the two binary input values, S is the sum, and C is the carry to the next position.

As its name suggests, the half adder is not a complete solution to the problem of binary addition. It works perfectly well for the last bit position in two binary numbers, but not for the rest of the bit positions because those positions may also involve a carry from the previous position. Incorporating the previous carry into the computation requires adding more logic gates to create a full adder, which looks like this:

![Full Adder Circuit Diagram]

In this diagram, Cin represents the carry from the previous position and Cout is the new carry passed along to the next adder. The full adder uses two XOR gates to produce the signal S and a majority circuit to produce the signal Cout. The operation of the full adder is summarized in the table in the right margin, which shows the value of the output signals for every possible input.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

You can use the adder circuits to add binary integers of any length. All you need to do is chain several full adders together, one for each bit in the internal binary representation of an integer. Figure 5-3 on the next page shows the complete implementation of a circuit that adds two eight-bit binary integers.
Figure 5-3: An eight-bit adder showing the sum of 42 and 57

\[
\begin{array}{c}
42 & 101010 \\
+57 & +111001 \\
99 & 110011 \\
\end{array}
\]
Figure 5-3 also shows how the circuit adds the numbers 42 and 57 by tracing the signals through all 40 of the gates in the circuit. That circuit would be almost impossible to understand if you looked only at the individual circuit elements. If you instead see it as eight one-bit adders, the complexity becomes much more manageable.

**Completeness of logical operators**

When Claude Shannon wrote his thesis exploring the link between mathematical logic and switching circuits, he began by showing how switches could implement the logical operations and, or, and not. This strategy made sense for two reasons. First, mathematicians had already established that these three operations are sufficient to compute any logical function by combining them in the appropriate way. Second, these three operations have natural counterparts in English, which makes them easy to understand.

Even though these three operations are ideal for humans, it does not necessarily follow that these operations are optimal to use in hardware design. The biggest problem is that there are three of them, which means that any fabrication technology that used these operations would need to be able to produce three different types of gates, which would almost certainly increase the cost.

It is easy to reduce the number of different gates to two, because it is possible to define the OR gate using the following combination of NOT and AND gates:

![Diagram of OR gate using NOT and AND gates]

This transformation circuit follows directly from the following logical rules, which are called *De Morgan’s laws* after the British mathematician Augustus De Morgan:

\[
\text{NOT (A AND B)} \text{ is equivalent to (NOT A) OR (NOT B)}
\]

\[
\text{NOT (A OR B)} \text{ is equivalent to (NOT A) AND (NOT B)}
\]

Sets of logical operations that can be combined to generate all possible logical functions are said to be complete, and the same terminology applies equally well to gates. The combination AND and NOT represents a complete set of logic gates from which all logical functions can be constructed. The combination OR and NOT works equally well. Thus, it is certainly possible to create digital logic using only two gate types.
The **NAND** and **NOR** gates

It is, however, possible to go one better. There are two logical functions, which correspond to the logic gates called **NAND** and **NOR**, either of which is complete entirely on its own. The symbols for these gates look like this:

As the symbols suggest, the **NAND** and **NOR** gates are simply the standard **AND** and **OR** gates with an inverter tacked on the end. The operator matrices for **NAND** and **NOR** appear, along with those for the gates you have already seen, in Figure 5-4.

To show that either of these gates is complete on its own, it is sufficient to show that you can string together combinations of this one gate type to implement a set of gates already known to be complete. To show that **NAND** is complete, for example, all you need to do is implement **NOT** and **AND**, like this:

The construction that uses **NOR** to implement **NOT** and **OR** is essentially the same.

The fact that either of these gates serves as a complete foundation for logic means that hardware designers can build arbitrarily large and complex circuits using only one type of gate. As a result, the tiny integrated circuits that power modern computers can be optimized so that they contain only **NAND** gates (or **NOR** gates depending on the fabrication technology), resulting in a significant savings in cost.

<table>
<thead>
<tr>
<th>name</th>
<th>symbol</th>
<th>table</th>
<th>name</th>
<th>symbol</th>
<th>table</th>
</tr>
</thead>
</table>
| NOT  | ![NOT symbol](image) | \[
|      |        | \begin{array}{c|c|c}
|      | A      | 0 & 1  \\
|      | NOT A  | 1 & 0  \\
| AND | ![AND symbol](image) | \[
|      | A      | 0 & 1  \\
|      | B      | 0 & 0  \\
|      | A AND B| 0 & 1  \\
| OR  | ![OR symbol](image) | \[
|      | A      | 0 & 1  \\
|      | B      | 0 & 1  \\
|      | A OR B | 0 & 1  \\
| XOR | ![XOR symbol](image) | \[
|      | A      | 0 & 1  \\
|      | B      | 0 & 0  \\
|      | X OR B | 0 & 1  \\
| NAND | ![NAND symbol](image) | \[
|      | A      | 0 & 1  \\
|      | B      | 1 & 1  \\
|      | NAND A | 1 & 0  \\
| NOR | ![NOR symbol](image) | \[
|      | A      | 0 & 1  \\
|      | B      | 0 & 0  \\
|      | A NOR B| 0 & 0  \\

F I G U R E  5-4 Standard logic gates
5.3 Memory

Although you have already seen how electronic circuits can be used to add binary numbers, you have not yet learned how to store those numbers in memory. Memory cells inside a computer support two operations. The read operation looks at the contents of a memory cell to determine what value is stored there. The write operation stores a new value in memory, overwriting whatever value was stored previously. The fundamental property of memory is persistence. If you write a value to a memory cell, that cell should store that value until some later write operation explicitly changes it.

The problem of storing a number can be simplified to the problem of storing a single bit because you can store large binary numbers by chaining individual bits together. A circuit that implements a single bit of memory must therefore support the following operations:

- **Setting a bit.** This operation sets the value of a bit to 1, which has the effect of turning it on.
- **Resetting a bit.** This operation sets the value of a bit to 0, which has the effect of turning it off. Computer scientists also call this operation clearing a bit.
- **Reading a bit.** The circuit must make it possible to determine whether a bit is on or off without permanently changing its value. Particularly in older memory technologies, reading a bit changes the value of that bit temporarily. In such cases, the hardware is responsible for restoring the state of the bit by rewriting its previous value.

The sections that follow illustrate how these operations can be implemented in hardware.

**Flip-flops**

As was the case with logic gates, it is easiest to understand memory if you take an abstract view that focuses on the behavior of the operations rather than their implementation. A single bit of memory is most commonly represented as a flip-flop, which is an abstract circuit element that can flip back and forth between two stable states. A flip-flop has two input signals and two output signals, as illustrated in the following “black box” diagram that hides the inner workings:
It is easiest to describe a flip-flop starting with the output signals. A flip-flop exists in two different states, which can be designated as on and off. The signal \(Q\) mirrors the state of the flip-flop as a whole. When the flip-flop is off, the output signal \(Q\) has the value 0; when it is on, \(Q\) has the value 1. The signal \(\bar{Q}\) carries the inverse of \(Q\). When the flip-flop is off, the output signal \(\bar{Q}\) has the value 1; when it is on, \(\bar{Q}\) has the value 0.

The input signal \(R\) represents the reset operation. Sending a 1 signal along the \(R\) input line turns the flip-flop off. The state of the flip-flop is recorded in its output signals. Sending the signal 1 along the \(R\) input line therefore leaves the flip-flop in the following state:

The input signal \(S\) represents the set operation. Sending a 1 signal along the \(S\) input line turns the flip-flop on, as follows:

In this configuration, the output signal \(Q\) has the value 1, and the output signal \(\bar{Q}\) has the value 0.

The utility of the flip-flop circuit arises from the fact that its state is persistent. Removing the 1 signal from the \(R\) or \(S\) input lines leaves the flip-flop unchanged, thereby allowing it to remember the state set by the last operation.

**Implementing a flip-flop using electromagnets**

One of the advantages of taking an abstract view of a circuit element is that you don’t need to worry about its implementation. This means that you can either ignore the implementation entirely or, alternatively, imagine an implementation that provides you with some useful intuition. It doesn’t matter if the circuit isn’t actually built that way because everything would work perfectly well if it were.
Figure 5-5 shows how engineers might have built a flip-flop in Claude Shannon’s day. The implementation uses two electromagnets to move a switch bar back and forth between two positions, as follows:

The gray areas of the switch bar represent some conducting material that responds to magnetic force, such as iron or steel. The white section in the middle of the switch bar is made from some insulating material. When the switch bar has been pulled toward the top as shown in the left-hand diagram, the contacts on the left side connect with the uppermost contacts in each pair on the right. When the switch bar has been pulled toward the bottom, the contacts on the left connect to the lowermost contacts on the opposite side of the bar. As you can see from Figure 5-5, this arrangement ensures that the output signals $Q$ and $\overline{Q}$ are connected to one of the $\oplus$ terminals, which represent the constant signals 1 and 0, respectively.

The flip-flop shown in Figure 5-5 works only if the switch bar stays put unless one of the electromagnets is active. That property ensures that the flip-flop state is persistent. This goal can be achieved by using a spring that pushes the bar to one side or the other, but with a weak force that the electromagnets can overcome.
Implementing a flip-flop using NOR gates

In practice, flip-flops are implemented using logic gates combined in a way that allows the circuit to exist in either of two stable states. One of the simplest designs for a flip-flop uses two NOR gates in the following arrangement:

Sending a 1 down the R input line turns the top NOR gate off, since the output of a NOR gate is 0 if either inputs is a 1. The bottom NOR gate, however, is on because both of its inputs are 0. The resulting circuit therefore looks like this:

The interesting feature of a flip-flop is that the output of each NOR gate is fed back into the input of the other. Doing so means that it is possible to turn off the signal on the R input line without affecting the state of the flip-flop. The top NOR gate stays off because of this feedback signal, as shown in the following diagram:

Sending a 1 down the S input line runs this same process in reverse. The bottom NOR turns off and the top NOR turns on, producing the following state:

Once again, removing the signal from the S input line has no effect.
5.4 Implementing digital logic

Logic gates can be implemented in many different ways. When Claude Shannon wrote his Master’s thesis in 1937, he envisioned building gates using relay circuits, and several early computers incorporated relays and similar electromechanical devices into their design. Computers built during World War II—including the Colossus machine that decoded the German diplomatic code that you will learn about in Chapter 11—replaced those relays with vacuum tubes that performed the same functions much more quickly. In the 1950s, vacuum tubes were superseded by transistors, which were smaller, cheaper, and more reliable. Modern computers still use transistors to implement their internal operations, but at a scale that would have been unimaginable to the team of inventors at Bell Labs who created the first transistor in 1947. With today’s technology, it is commonplace to build single microprocessor chips that contain billions of individual transistors, wired together to implement gates, adders, and all the other functional units that enable computation.

Semiconductors

Transistors operate by exploiting the electrical properties of semiconductors, which are materials whose resistance falls at an intermediate point on the spectrum between conductors that allow current to flow freely and insulators that block it. Semiconductors also change their conductivity in response to several factors, including the presence of impurities in the semiconducting material and the effect of electric charges. These properties make it possible to use semiconducting materials as a switch. Forcing the semiconductor into a state where it acts mostly as an insulator turns the switch off; forcing it to act as a conductor turns the switch on.

Although some compounds also exhibit semiconducting properties, the most common materials used commercially as semiconductors are the elements silicon and germanium. These elements, whose chemical symbols are Si and Ge, appear in the same chemical group as carbon, as shown in the following diagram of the first four rows of the periodic table:

```
H          He
Li   Be
Na   Mg
K   Ca   Sc   Ti   V   Cr   Mn   Fe   Co   Ni   Cu   Zn   Ga   Ge   As   Se   Br   Kr
B   C   N   O   F   Ne
Al   Si   P   S   Cl   Ar
```

The elements in this group have four electrons in their outer shell, which makes the shell exactly half full. When these elements enter into chemical reactions, they are equally likely to give up four electrons as to acquire four from their neighbors. This flexibility makes these elements particularly sensitive to the presence or absence of
free electrons in the crystal lattice, which in turn makes circuits created from these materials easier to control.

In its pure crystalline form, elemental silicon forms a tight crystal lattice with the same structure that carbon uses to form diamonds. The rigid structure of that lattice makes it hard for electrons to move, which means that it acts like an insulator. Adding impurities to the crystal lattice changes that conductivity. That process is called **doping**. Doping a semiconductor with an element that appears in the column to the right of silicon in the periodic table, such as phosphorus, introduces free electrons into the crystal lattice. Doping a semiconductor with an element that appears in the column to the left, such as boron, leaves spaces for electrons in the lattice. Those spaces are called **holes**. Because electrons are negatively charged, semiconductors with extra electrons are called **n-type semiconductors**, while those with holes are called **p-type semiconductors**. Transistors work by taking advantage of the fact that the resistance across the boundary of an n-type and a p-type semiconductor can be controlled by the presence or absence of an electric charge.

**Field-effect transistors**

Most transistors used in computing hardware are **field-effect transistors** (usually referred to by the acronym **FET**), which rely on a gate element to create the electric field that controls the conductivity across the transistor as a whole. Field-effect transistors come in two types, which differ in the internal arrangement of n-type and p-type semiconductors. In cross-section, the two types of FET look like this:

Both types of FET have three leads, which are called **source**, **gate**, and **drain**. Changing the voltage on the gate affects the resistance between the source and the drain, which controls the flow of current. In the n-channel FET, putting a positive charge on the gate attracts free electrons from the n-type regions, creating a channel that allows current to flow. In the p-channel FET, the charge on the gate is adjusted so that it ordinarily contains a negative charge, which forms a conductive channel composed of holes taken from the p-type regions. Putting a positive charge on the gate drives the holes away, blocking the current. The effect of putting a positive charge on the gate of each of these FETs is illustrated in the following diagrams:
As these diagrams illustrate, FETs implement two types of electronic switches. Placing a positive charge on the gate of an n-channel FET turns the switch on; placing a positive charge on the gate of a p-channel FET turns the switch off.

Field-effect transistors can be incorporated into logic diagrams, in which they are represented by the following symbols:

![NFET symbol](image1)

![PFET symbol](image2)

To maintain consistency with the typographical style of logic gates, it makes sense to abbreviate the names to **NFET** and **PFET** and to set those names in bold, sans-serif type. The circle in the **PFET** symbol indicates inversion of the signal, just as it does in the **NOT**, **NOR**, and **NAND** gates.

**CMOS logic**

Field-effect transistors can be used to implement digital logic in a variety of ways. One of the most widespread techniques for building digital logic is called **CMOS**, which stands for **complementary metal-oxide semiconductor**. The second part of the name refers to the structure of the field-effect transistor itself, in which the metal of the gate is separated from the underlying semiconductor by an insulating layer usually composed of silicon dioxide (SiO₂), which is the mineral quartz. The word *complementary* refers to strategy of using transistors in pairs, so that each **NFET** is balanced by a **PFET** that ensures that every wire in the circuit is connected to one of the standard voltages, which are indicated in this book by the symbols ⊕ and ⊖. For historical reasons, electrical engineers refer to these constant signals using the names **Vcc** (for voltage at the common collector) and **GND** (for ground). These names, however, only add to the confusion. In these diagrams, the ⊖ symbol indicates a positive voltage that corresponds to the binary digit 1. The ⊕ symbol corresponds to the binary digit 0.

The idea of complementarity is easily illustrated by the **NOT** gate, which looks like this when implemented in CMOS, for each possible value of the signal A:
The use of complementary PFET and NFET circuits ensures that the output is connected directly either to the terminal or the terminal, so that it is guaranteed to have the value 0 or 1. Figure 5-6 illustrates the implementation of the NAND gate in CMOS, mapping the signals for all four possibilities of the inputs A and B.

**Very large scale integration**

The NAND gate shown in Figure 5-6 makes it possible to implement any logical function by wiring NAND gates together in the appropriate ways. A computer chip does just that on an enormous scale. The details of the process are far too complex to include in this book, but Figure 5-7 on the next page offers an overview of the steps involved in creating chips that contain millions or even billions of individual field-effect transistors. The process of fabricating chips with such large transistor counts is called very large scale integration, or VLSI for short.
**Figure 5.7** Steps involved in chip fabrication

Step 1: Start by growing an ultrapure silicon cylinder.

Step 2: Slice the cylinder into thin wafers.

Step 3: Oxidize the surface to create an SiO$_2$ layer.

Step 4: Add a light-sensitive layer called **photoresist**.

Step 5: Project an image of the circuit through a mask.

Step 6: Repeat step 5 to create additional circuits.

Step 7: Etch away the exposed photoresist and SiO$_2$.

Step 8: Remove any remaining photoresist.

Step 9: Use an ion gun to create n- and p-type regions.

Step 10: Add insulators and metal conductors.

Step 11: Separate the chips with a diamond saw.

Step 12: Package and test the individual chips.
In Babbage’s Analytical Engine and in several of the earliest computers, programs and data were rigidly separated. Babbage envisioned a machine in which the program would be encoded on punched cards similar to those used in the Jacquard Loom of the early 19th century. In Howard Aiken’s Mark I described on page 72, programs were fed into the machine on paper tape. In each of those designs, the numeric values manipulated by a program were stored in an entirely different form.

One of the most important breakthroughs in the history of computing was the realization that programs and data could be stored in the same memory, as long as the programs could be represented in numeric form. Fortunately, creating a numeric representation for programs is no harder than creating a numeric representation for characters, as described in Chapter 4. If a computer uses a particular set of instructions, its designers could apply the principle of enumeration by assigning a number to each instruction in some order. That number, together with additional numeric information to indicate what memory location was involved, would then constitute a machine instruction. A program is then simply a sequence of machine instructions, each of which has a numerical value. The technique of storing programs and data in the same memory is called the stored-programming model.

It is impossible—and presumably unnecessary—to identify a single inventor of the stored-programming idea. This design is traditionally called the von Neumann architecture after the Hungarian/American mathematician John von Neumann, who published a detailed description of the model in 1945. At that time, however, the stored-program idea was already circulating among American computing pioneers. The fundamental ideas are even older than that. Representing programs as data is central to Alan Turing’s work on computability, as discussed in Chapter 8. That idea is also used in computers built by German engineer Konrad Zuse in the 1930s.
The first stored-program computer

Over the years, the question of who deserves the credit for inventing the computer has generated considerable controversy. In the United States, patent rights for the digital computer were originally awarded to John W. Mauchly and J. Presper Eckert, who designed and built the Electronic Numerical Integrator and Computer, more commonly known as ENIAC. In 1973, a judge for the Federal District Court of Minnesota invalidated that patent on the grounds that the ENIAC incorporated several features from an earlier computer designed at Iowa State College (now Iowa State University) by John Atanasoff and his graduate student Clifford Barry.

As you will discover in this chapter, it is equally difficult to assign credit for the stored-programming concept that forms the central theme of this chapter. Although the idea is traditionally credited to the mathematician John von Neumann, many other people were also central to its development.

There is no such controversy, however, as to when the stored-programming model was first implemented in practice. That distinction belongs to the Manchester Small-Scale Experimental Machine, nicknamed the Baby, which ran its first program on June 21, 1948.

The leaders of the design team at the University of Manchester were Frederic C. Williams, Tom Kilburn, and Geoff Tootill. The primary goal of the project was to demonstrate the feasibility of the stored-program model rather than to build a computer that could be used for practical applications. As a prototype, however, the Baby was extremely successful and led directly to the development of the Ferranti Mark I, which was the first commercial computer to use the stored-program design.

Despite its nickname, the Baby was gargantuan by modern standards, at least in terms of its physical dimensions. It stood more than seven feet tall and occupied a series of cabinets that extended for 17 feet. At the same time, the computational power of the Baby was indeed minuscule. Modern processors have billions of times the capacity of the Baby, even though they fit on a silicon chip smaller than a penny.

In 1998, computer history enthusiasts from the British Computer Society created a replica of the Baby to celebrate its fiftieth anniversary. That replica is now on display at the Museum of Science and Industry in Manchester, England.
The first computer to use the stored-program architecture was the Manchester Small-Scale Experimental Machine, better known by its nickname Baby, which is described on the opposite page. The Baby was designed as a proof-of-concept experiment that would demonstrate the feasibility of both the stored-program model and of the various hardware devices used to implement it. Its capabilities were severely limited. The Baby’s repertoire of seven instructions included subtraction but not addition, because it is possible to simulate addition by subtracting a number from zero and then subtracting that result from some other value. The Baby offered only 32 words of memory, which had to store both the data and the program instructions. These restrictions rendered the Baby unsuitable for anything other than the simplest of programs, such as carrying out a single long division, finding the largest factor of a number, or calculating the greatest common divisor.

6.1 Introducing Toddler

The restrictions on the Baby are so severe that it doesn’t work well as a teaching tool. To give you a sense of how von Neumann architectures work, this chapter introduces a more powerful computer that is a few steps up from the Manchester Baby, which I’ve chosen to call Toddler. Although Toddler’s increased size and power makes it possible to write more interesting programs, the machine is still small enough so that you can see its entire structure in a single diagram, as shown in Figure 6-1.

The structure of the Toddler machine

Like Babbage’s Analytical Engine, Toddler is divided into two parts. The left side of the diagram in Figure 6-1 shows the components of the central processing unit,
or CPU, which acts as the computational engine for the machine. The right side of
the diagram shows the memory, which consists of 100 words, where each word
contains three decimal digits preceded by a plus or a minus sign.

After spending an entire chapter extolling the virtues of the binary system, the
choice of decimal representation for the Toddler machine will certainly seem like a
step backward. As it happens, Howard Aiken was partly right in his defense of the
decimal system: decimal representation is, undeniably, easier for humans to
understand than its binary counterpart. Given that the goal of the Toddler machine
is to help you understand how computers work on the inside, it makes sense to use
decimal arithmetic, at least at the beginning. The Toddler simulator also runs in
hexadecimal mode, and you can play around with that if you want additional
practice with binary arithmetic.

Each individual word in Toddler’s memory is identified by a numeric address,
which ranges from 00 to 99. Because doing so makes it easier to catch various
common programming errors, the Toddler hardware makes it illegal to use address
00, which therefore appears as a gray box in Figure 6-1. This design decision,
which is replicated in some form in most modern architectures, means that the
usable memory actually begins at address 01.

Given that each of Toddler’s memory words holds a three-digit signed integer,
the range of each memory word is –999 to +999. The following diagram therefore
shows a possible configuration of Toddler’s memory, although only a few words of
the 99 usable words are shown:

```
+8 5 0
+8 5 1
+1 5 0
  ...
-0 2 3
+0 6 5
  ...
+0 0 0
+0 0 0
```

In the diagram, the word at address 01 contains the number +850, the word at
address 50 contains the number –23, and so forth.

Even though memory locations in the Toddler machine contain three-digit
signed integers, it is important to keep in mind that you can interpret those integers
in different ways. For example, the value 65 in location 51 could represent either
the number 65 or the Unicode character ’A’, as described in Chapter 4. Both the
number and the character have the internal representation 65. The correct interpretation depends on how the value is used.

In addition to the circuitry required to implement the operations of the machine, the CPU includes several specialized memory cells called **registers**. Toddler has the following registers:

- **AC**: The accumulator, which is used for arithmetic computation
- **PC**: The program counter, which determines which instruction to execute
- **IR**: The instruction register, which decodes the current instruction
- **XR**: The index register, which is used to simplify operations on arrays
- **SP**: The stack pointer, which is used to keep track of function calls

The purpose and format of each of these registers will be introduced as it becomes relevant, which will in some cases not happen until a later chapter.

### 6.2 Programming the Toddler machine

In order to represent instructions inside a machine, there needs to be some encoding scheme that allows the hardware to interpret the contents of some memory location as an executable instruction. In the Toddler machine, each instruction is divided into two parts. The first digit of the memory word—along with the sign—indicates what instruction is being performed, and the last two digits indicate an address in memory. Thus, an instruction in Toddler is logically divided into the following fields, which are called the **opcode** (short for operation code) and the **address**:

```
  opcode  address
```

For example, if Toddler were to execute the value in address 01 above as an instruction, it would take the value +850 and divide it into its opcode and address components, as follows:

```
+ 8 5 0
```

The first digit (together with the sign, which is used for extended instructions described later in this book) specifies the code for the particular instruction to be performed. The last two digits specify the address of the word in memory on which the operation will be performed.

**The Toddler instruction set**

The standard instructions of the Toddler machine consists of the nine instructions shown in Figure 6-2. In the current example, the +8 opcode specifies an **INPUT**
instruction, which reads in a value from the user. Given that the complete instruction word is +850, the value entered by the user is stored at address 50.

You can use these instructions to write simple programs for Toddler that are in many ways similar to the programs for the Analytical Engine from Chapter 3. The following program, for example, reads in two numbers from the user and then prints their sum:

+850
+851
+150
+351
+252
+952
+500

By convention, the instructions in a Toddler program are stored in memory, beginning at address 01. This program therefore occupies the first seven words of memory, as follows:
The first line of the program is the instruction +850, which represents an **INPUT** instruction. When the Toddler machine encounters this instruction, it types out a question mark and then waits for the user to enter a number. When the user hits the **RETURN** key to signal the end of the number, the Toddler machine stores that value in location 50. The next instruction does the same thing for the second input value, storing the result in location 51. In this program, memory locations 50 and 51 are used to hold data values, which are being interpreted here as integers. Locations used to hold data that changes over the course of the program are called **variables**.

At this point in the execution of the program, the variables in location 50 and 51 contain the two input values. The next step in the process is to add the numbers together. In the Toddler machine, all arithmetic must be done in the **AC** register. Thus, to add the two numbers, the program must first load one of the values into the **AC**, add the second, and then store the result back into memory—much as in Babbage’s Analytical Engine. These operations are accomplished using the following code, which consists of a **LOAD** instruction, an **ADD** instruction, and a **STORE** instruction:

```
+150
+351
+252
```

At this point, address 52 contains the desired result. To display that result back to the user, the program executes the +952 instruction in location 06, which is an **OUTPUT** instruction. The program then moves on to the +500 at location 07, which is a **HALT** instruction. At the end of the program, the Toddler console looks like this:

```
<table>
<thead>
<tr>
<th>Toddler Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>? 25</td>
</tr>
<tr>
<td>? 17</td>
</tr>
<tr>
<td>42</td>
</tr>
</tbody>
</table>
```

**The instruction cycle**

By convention, all Toddler programs begin at address 01. To make sure that instructions are executed in their proper order, Toddler—like any machine that uses the von Neumann architecture—devotes an internal register that keeps track of the next instruction in sequence. That register is called the **program counter** or **PC**. When the program is started, the **PC** is set to 01 to indicate that the first instruction to be executed comes from address 01. Toddler uses another internal register, called the **instruction register** or **IR**, to hold the three-digit instruction word.
For each instruction, Toddler executes the following instruction cycle:

1. **Fetch the current instruction.** In this phase of the instruction, Toddler finds the word from the memory address specified by the PC and copies its value into the IR.

2. **Increment the program counter.** Once the current instruction has been copied into the IR, Toddler adds one to the PC so that it indicates the next instruction in sequence.

3. **Decode the instruction in the instruction register.** The value copied into the IR is a three-digit integer. To use it as an instruction, Toddler must divide the instruction word into its opcode and address components.

4. **Execute the instruction.** Once the operation code and address field have been identified, the Toddler processor must carry out the steps necessary to perform the indicated action.

This cycle is repeated until a HALT instruction is executed or an error occurs.

**Controlling the order of execution**

The add-two-numbers program on page 124 runs through its instructions in a completely linear fashion, starting at the beginning and moving forward until it reaches the end. In order to perform more complex calculations, programs for the Toddler machine must be able to control the order in which instructions are executed. To do so, Toddler uses the JUMP instructions, which come in three forms. The JUMP instruction itself jumps unconditionally to the address stored in the instruction. The JUMPZ and JUMPN instructions are similar, but jump to the address only if a particular condition is met. The JUMPZ instruction jumps only if the value of the AC is zero; the JUMPN instruction jumps only if the value of the AC is negative.

As an example, suppose that you want to write a program that adds numbers entered by the user until the user enters 0 to mark the end of the input. In English, you can express the logic of such a program as the following series of steps:

1. Designate a memory location called total to record the total so far.
2. Designate a memory location called value to hold each value as it appears.
3. Initialize total to zero.
4. Use the INPUT instruction to read a number into value.
5. If value is zero, output the value in total and halt.
6. Add value to the contents of total.
7. Go back to step 4 to get another number.
In Toddler, you can easily write a simple program to execute this series of steps. That program looks like this in its purely numeric form:

```
01  + 1 1 1
02  + 2 1 2
03  + 8 1 3
04  + 1 1 3
05  + 6 0 9
06  + 3 1 2
07  + 2 1 2
08  + 5 0 3
09  + 9 1 2
10  + 5 0 0
11  + 0 0 0
12  + 0 0 0
13  + 0 0 0
```

This program uses a +609 instruction at address 05 to check whether the user entered a zero value. If so, the program jumps ahead to print out the result. It also uses a +503 instruction at address 08 to jump back to the instruction at address 03 that reads in the next value.

### 6.3 Assembly language

Although it is possible to trace the operation of the program to add a list of numbers in its numeric form, it is tedious to do so. While numeric representation makes it easier to store instructions in memory, it also makes it harder for programmers to understand what is going on. From the earliest days of computing, stretching back to Ada Lovelace’s notes on the Analytical Engine, programmers have preferred to write instructions in a more human-readable form. Programs written in their numeric form are called machine language programs. Programs that use symbolic instruction names in place of the numeric opcodes are referred to as assembly language programs. Assembly language programs are much easier to read, but can nonetheless be translated directly into their machine language counterparts. An assembly language version of the AddList.asm program appears in Figure 6-3.

The most important assembly-language feature introduced in Figure 6-3 is the use of instruction names to represent particular opcodes. This makes it possible, for example, to write

```
HALT
```
at the end of the program in preference to the machine-language instruction

\[ +500 \]

The second feature introduced in Figure 6-3 is the use of symbolic names—such as `start`, `loop`, `done`, `zero`, `total`, and `n`—to refer to specific addresses in the program. In assembly language, such names are known as *labels*. Labels in Toddler are defined by writing a name followed by a colon, which defines that name as being equal to the current location in memory. For example, the first line defines the label `start` to have the value 1, since this instruction is being placed in location 01. Similarly, the symbol `loop` will have the value 3.

Labels may be used before they are defined; when the actual definition appears, the appropriate value is substituted back into any instructions that use it. Thus, when Toddler gets to the line labeled `zero` at memory location 11, it not only defines the label `zero` to have the value 11, but also goes back and fills in 11 as the address part of the instruction

\[ \text{LOAD} \quad \text{zero} \]

in memory location 01. Because `LOAD` has the operation code +1, the value of memory location 01 after loading the assembly language version of the program will be +111, just as it appears in the machine-language version.
Another important concept to take from Figure 6-3 is how to specify constant values in a Toddler program. In the English version of the program, the first operation after giving names to the data values is to set the variable total to zero. To do so, you could not simply write

```
LOAD  0
STORE total
```

The LOAD instruction here will try to load the value in address 0, rather than the integer value 0, which is what the program needs. To work with a constant integer value, you need to put that constant in a memory word and then specify its address in the appropriate instruction. Here, for example, the instruction

```
LOAD    zero
```

loads from the address corresponding to the label zero, which is defined by the program to contain the value 0, as follows:

```
zero:   0
```

Because it is cumbersome to define all constants by putting them in a memory word and then using that address in other instructions, the Toddler assembler allows you to specify constants by writing a number sign (#) before an integer value, as in

```
LOAD    #0
```

What the assembler does when it encounters the number sign is

1. Find some unused address at the end of the program.
2. Put the constant value into that address.
3. Use the address of the constant in the instruction that contained the constant.

The # syntax therefore has exactly the same effect as storing the constant in a memory location and using that location’s address. The advantage of using the # form is that the resulting program is easier to read.

**Playing computer**

To get a better sense of how von Neumann machines work, it is important to go through the operation of a program on your own to make sure you can execute all the machine instructions. As an example, follow through the logic of the AddList.asm program from Figure 6-3 using the input values 1, 2, 3, 4, and 0. The output of that program looks like this:
The important thing here is to understand how the instructions in the program accomplish the task.

(a) Work through the Toddler program shown in the memory diagram in Figure 6-1. What value does the program compute, assuming that the input values are positive integers.

(b) Write a Toddler program that displays the squares of the integers between 0 and 31. The logic of the program is similar to the program in Figure 3-5, which solves the same problem for Babbage’s Analytical Engine.

(c) Write a Toddler program called Largest.asm that allows the user to enter a list of integers and then reports the largest value. As in the AddList.asm program, the user indicates the end of the list by entering a zero value. The output of the program is illustrated by the following sample run:

In writing the program, you may assume that the user enters at least one data value before entering the sentinel, but you may not assume that the input values are always positive.
When you write a program to solve a particular problem, you can usually choose from many different solution strategies. Your first consideration in choosing a particular strategy is to make sure that the solution is effective. If your program doesn’t produce the right answer, you need to find an algorithm that does.

Within the space of effective algorithmic solutions, however, you still have to make choices. Some algorithms, for example, are simpler than others. As a general rule, it makes sense to choose simple algorithms whenever you can. Programs that rely on simple algorithms are easier to write, debug, and maintain. Given that software development and maintenance are expensive undertakings, reducing complexity usually reduces the overall cost. All too often, programmers try to be overly clever, adding complexity to their code even when doing so offers no significant advantage.

There are, however, situations in which it is critical to think about efficiency. Shaving five or ten percent off the running time of a program is rarely worth the effort unless you are working for a company that needs to have the fastest application possible. By contrast, if you can reduce the running time of your program by a factor of a thousand or more, it certainly seems worth considering an algorithm that offers that kind of efficiency gain.

Before you can learn how to make choices among competing algorithms, it is important to ask a few questions. What does the term efficiency mean in an algorithmic context? How would you go about measuring that efficiency? These questions form the foundation for the subfield of computer science known as analysis of algorithms.
Donald Knuth and The Art of Computer Programming

Those of us who studied computer science in the 1970s (and for many years beyond that) learned about algorithms from a remarkable set of books by Don Knuth, who is now Professor Emeritus of the Art of Computer Programming at Stanford University. The first three volumes—each of which has been revised over time and which were recently joined by the first part of a much anticipated fourth volume—cover the science of algorithms with such thoroughness and detail that The New York Times once described them as “the profession’s founding treatise.”

Professor Knuth has long argued that computer science, despite its name, is more art than science. Throughout his career, Knuth has championed the practice of that art, weaving the notions of aesthetics and elegance into the practice of computing. When he became convinced that conventional typesetting was unable to produce books that would be beautiful as well as comprehensive, Knuth implemented the typesetting language TeX, which remains in widespread use today. Along the way, he worked with artists and font designers to create METAFONT, which allows its users to create new fonts using a mathematical model inspired by the brush strokes used in calligraphy. His excursions into the world of typesetting and calligraphy also enabled him to produce a compendium of Biblical verses entitled 3:16, illustrated by calligraphic artists from around the world. His reverence for aesthetics has led Knuth to play the organ, write a mathematical novel (Surreal Numbers), and use language in a way that sparkles with brilliance and life.

Professor Knuth received the ACM Turing Award in 1974 for his contributions to computer science. In his Turing Award lecture, excerpted below, Knuth defends his claim that computer programming is an art that produces objects of beauty.

Knuth is by no means alone in his belief that computing and artistry are inextricably linked. The late Steve Jobs, co-founder and longtime CEO of Apple Computer, Inc., told the New York Times in 1997 that “the Macintosh turned out so well because the people working on it were musicians, artists, poets, and historians who also happened to be excellent computer scientists.”

When Communications of the ACM began publication in 1959, the members of ACM’s Editorial Board made the following remark as they described the purposes of ACM’s periodicals: “If computer programming is to become an important part of computer research and development, a transition of programming from an art to a disciplined science must be effected.” Such a goal has been a continually recurring theme during the ensuing years; for example, we read in 1970 of the “first steps toward transforming the art of programming into a science.” Meanwhile we have actually succeeded in making our discipline a science, and in a remarkably simple way: merely by deciding to call it “computer science.”

Implicit in these remarks is the notion that there is something undesirable about an area of human activity that is classified as an “art”; it has to be a science before it has any real stature. On the other hand, I have been working for more than 12 years on a series of books called “The Art of Computer Programming.” People frequently ask me why I picked such a title; and in fact some people apparently don’t believe that I really did so, since I’ve seen at least one bibliographic reference to some books called “The Act of Computer Programming.” . . .

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the state of the Art.

—Don Knuth, Turing Award Lecture, 1974
Although a detailed account of algorithmic analysis is beyond the scope of this book, you can get a sense of the field by investigating the performance of a few simple algorithms. This chapter begins that investigation by looking at the following operations on array data:

- **Searching**, which consists of finding a particular value in an array. In computer science, the value you are searching for is often called the *key*.
- **Sorting**, which consists of rearranging the elements of an array into some predefined order.

As you will discover, there are many algorithms for solving each of these problems. These algorithms, moreover, vary enormously in their efficiency. Given that both searching and sorting come up frequently in practice, choosing an efficient algorithm turns out to be extremely important in commercial applications.

### 7.1 Searching

As with most new concepts, the idea of algorithmic efficiency is easiest to present in the context of a specific example. On the one hand, the example must be small enough so that the details of the algorithms are easy to follow. On the other hand, the example must be large enough to illustrate how different algorithmic choices affect the efficiency of the solution. Throughout the discussion of searching that follows, the examples rely on an array consisting of 50 elements, one for each of the fifty U.S. states. The values in the array are the two-character abbreviations introduced by the United States Postal Service in 1963, as shown in Figure 7-1.

Typically, functions that search for a specific value in an array return the index at which the key appears, or some sentinel value if the key does not occur at all. For the purposes of this chapter, however, it makes sense to simplify the problem so that the goal is to determine whether or not a particular key value appears. If you translate that formulation into JavaScript, your task is to implement the function

```
function existsIn(key, array)
```

**Figure 7-1** The array of two-letter state codes

<table>
<thead>
<tr>
<th>STATE CODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>AK AL AR AZ CA CO CT DE FL GA HI IA ID IL IN KS KY</td>
</tr>
<tr>
<td>IA MA MD ME MI MN MO MS MT NC ND NE NH NJ NM NV NY</td>
</tr>
<tr>
<td>OH OK OR PA RI SC SD TN TX UT VA VT WA WI WV WY</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td>17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33</td>
</tr>
<tr>
<td>34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49</td>
</tr>
</tbody>
</table>
that returns `true` if the specified key value exists in the array and `false` if it does not. Thus, given the value of `STATE_CODES` from Figure 7-1, calling

```javascript
existsIn("NV", STATE_CODES)
```

should return `true`. Calling `existsIn` with any abbreviation that does not appear in the array should return `false`.

**Linear search**

When you need to find an element in an array, the simplest strategy—although not necessarily the most efficient one—is captured in the following advice that the King of Hearts gives the White Rabbit in Lewis Carroll’s *Alice’s Adventures in Wonderland*:

> Begin at the beginning, and go on till you come to the end: then stop.

Turning that informal statement into an algorithm for searching is not at all difficult. The only modification that you need to make is that the algorithm should also stop if it finds the key value. Thus, you might express a more complete account of the searching example algorithm as follows:

```javascript
function existsIn(key, array) {
    for (var i = 0; i < array.length; i++) {
        if (key === array[i]) return true;
    }
    return false;
}
```

The `for` loop begins at the beginning and continues until it comes to the end of the array. The function returns whenever it finds the key along the way or, failing that, at the end of the entire loop.

As long as the size of the array remains reasonably small, the linear search algorithm is almost certainly the best strategy for finding a specific element. It is easy to code, to the point that experienced programmers can write the necessary implementation without any serious thought. If the size of the array becomes larger, however, the algorithmic performance of linear search goes down. In the example
using the \texttt{STATE\_CODES} array, the linear search algorithm never has to look at more than 50 elements, which would take less than a microsecond on a modern computer—a speed so fast that it would seem instantaneous to any human observer. If, however, you were trying to use this strategy with an array containing all ten-digit telephone numbers in the United States, using linear search would require scanning through as many as ten billion elements. At that scale, the process slows down to the point that it is worth looking for a better strategy.

**Binary search**

In this example, it is possible to implement a more efficient strategy by taking advantage of the fact that the elements of the \texttt{STATE\_CODES} array appear in alphabetical order. The fundamental idea is to start the search, not at the beginning of the array, but in the middle. Given that there are 50 states with index values between 0 and 49, the middle element is the one that occurs closest to the average of those two indices, which is the element at index position 24:

If you’re looking for the two-letter code MS, you’re finished. If not, you can check to see whether the key you’re searching for comes before or after MS in alphabetical order. In either case, you can immediately rule out half the values in the array.

For example, suppose that you’re looking for the key OK—an abbreviation that has been made famous by the inclusion of the phrase “Oklahoma is OK” on the state’s license plates. The abbreviation OK comes after MS, so you immediately know that OK must come in the second half of the array. You can therefore eliminate the elements from 0 to 24 inclusive, leaving the following possibilities:

Once you have eliminated half of the original array, you can apply the same strategy to the remaining elements. This time, you need to look at the element that occurs midway between index positions 25 and 49, which is the element at index position 37. If you repeatedly apply this strategy looking for the code OK, you get the sequence of operations shown in Figure 7-2. Because this process divides the array into two pieces on each cycle, this algorithm is called \textit{binary search}. 
**Figure 7-2** Steps required to find OK using the binary search algorithm

Step 1. Look at the element in the middle of the array.

<table>
<thead>
<tr>
<th>AK</th>
<th>AL</th>
<th>AR</th>
<th>AZ</th>
<th>CA</th>
<th>CO</th>
<th>CT</th>
<th>DE</th>
<th>FL</th>
<th>GA</th>
<th>HI</th>
<th>IA</th>
<th>ID</th>
<th>IL</th>
<th>IN</th>
<th>KS</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>LA</td>
<td>MA</td>
<td>MD</td>
<td>ME</td>
<td>MI</td>
<td>MN</td>
<td>MO</td>
<td>(MS)</td>
<td>MT</td>
<td>NC</td>
<td>ND</td>
<td>NE</td>
<td>NH</td>
<td>NJ</td>
<td>NM</td>
<td>NV</td>
<td>NY</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>OH</td>
<td>OK</td>
<td>OR</td>
<td>PA</td>
<td>RI</td>
<td>SC</td>
<td>SD</td>
<td>TN</td>
<td>TX</td>
<td>UT</td>
<td>VA</td>
<td>VT</td>
<td>WA</td>
<td>WI</td>
<td>WV</td>
<td>WY</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>

OK comes after MS, so you can ignore the first half of the subarray.

Step 2. Look at the element in the middle of the remaining subarray.

<table>
<thead>
<tr>
<th>AK</th>
<th>AL</th>
<th>AR</th>
<th>AZ</th>
<th>CA</th>
<th>CO</th>
<th>CT</th>
<th>DE</th>
<th>FL</th>
<th>GA</th>
<th>HI</th>
<th>IA</th>
<th>ID</th>
<th>IL</th>
<th>IN</th>
<th>KS</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>LA</td>
<td>MA</td>
<td>MD</td>
<td>ME</td>
<td>MI</td>
<td>MN</td>
<td>MO</td>
<td>MS</td>
<td>MT</td>
<td>NC</td>
<td>ND</td>
<td>NE</td>
<td>NH</td>
<td>NJ</td>
<td>NM</td>
<td>NV</td>
<td>NY</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>OH</td>
<td>OK</td>
<td>OR</td>
<td>(PA)</td>
<td>RI</td>
<td>SC</td>
<td>SD</td>
<td>TN</td>
<td>TX</td>
<td>UT</td>
<td>VA</td>
<td>VT</td>
<td>WA</td>
<td>WI</td>
<td>WV</td>
<td>WY</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>

OK comes before PA, so you can ignore the second half of the subarray.

Step 3. Look at the element in the middle of the remaining subarray.

<table>
<thead>
<tr>
<th>AK</th>
<th>AL</th>
<th>AR</th>
<th>AZ</th>
<th>CA</th>
<th>CO</th>
<th>CT</th>
<th>DE</th>
<th>FL</th>
<th>GA</th>
<th>HI</th>
<th>IA</th>
<th>ID</th>
<th>IL</th>
<th>IN</th>
<th>KS</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>LA</td>
<td>MA</td>
<td>MD</td>
<td>ME</td>
<td>MI</td>
<td>MN</td>
<td>MO</td>
<td>MS</td>
<td>MT</td>
<td>NC</td>
<td>ND</td>
<td>NE</td>
<td>NH</td>
<td>NJ</td>
<td>NM</td>
<td>NV</td>
<td>NY</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>OH</td>
<td>OK</td>
<td>OR</td>
<td>PA</td>
<td>RI</td>
<td>SC</td>
<td>SD</td>
<td>TN</td>
<td>TX</td>
<td>UT</td>
<td>VA</td>
<td>VT</td>
<td>WA</td>
<td>WI</td>
<td>WV</td>
<td>WY</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>

OK comes after NJ, so you can ignore the first half of the subarray.

Step 4. Look at the element in the middle of the remaining subarray.

<table>
<thead>
<tr>
<th>AK</th>
<th>AL</th>
<th>AR</th>
<th>AZ</th>
<th>CA</th>
<th>CO</th>
<th>CT</th>
<th>DE</th>
<th>FL</th>
<th>GA</th>
<th>HI</th>
<th>IA</th>
<th>ID</th>
<th>IL</th>
<th>IN</th>
<th>KS</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>LA</td>
<td>MA</td>
<td>MD</td>
<td>ME</td>
<td>MI</td>
<td>MN</td>
<td>MO</td>
<td>MS</td>
<td>MT</td>
<td>NC</td>
<td>ND</td>
<td>NE</td>
<td>NH</td>
<td>NJ</td>
<td>NM</td>
<td>NV</td>
<td>NY</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>OH</td>
<td>OK</td>
<td>OR</td>
<td>PA</td>
<td>RI</td>
<td>SC</td>
<td>SD</td>
<td>TN</td>
<td>TX</td>
<td>UT</td>
<td>VA</td>
<td>VT</td>
<td>WA</td>
<td>WI</td>
<td>WV</td>
<td>WY</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>

OK comes after NY, so you can ignore the first half of the subarray.

Step 5. Look at the element in the middle of the remaining subarray.

<table>
<thead>
<tr>
<th>AK</th>
<th>AL</th>
<th>AR</th>
<th>AZ</th>
<th>CA</th>
<th>CO</th>
<th>CT</th>
<th>DE</th>
<th>FL</th>
<th>GA</th>
<th>HI</th>
<th>IA</th>
<th>ID</th>
<th>IL</th>
<th>IN</th>
<th>KS</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>LA</td>
<td>MA</td>
<td>MD</td>
<td>ME</td>
<td>MI</td>
<td>MN</td>
<td>MO</td>
<td>MS</td>
<td>MT</td>
<td>NC</td>
<td>ND</td>
<td>NE</td>
<td>NH</td>
<td>NJ</td>
<td>NM</td>
<td>NV</td>
<td>NY</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>OH</td>
<td>(OK)</td>
<td>OR</td>
<td>PA</td>
<td>RI</td>
<td>SC</td>
<td>SD</td>
<td>TN</td>
<td>TX</td>
<td>UT</td>
<td>VA</td>
<td>VT</td>
<td>WA</td>
<td>WI</td>
<td>WV</td>
<td>WY</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>

The circled element is equal to OK, so the search is finished.
Writing the code to implement binary search is certainly more complicated than implementing linear search, but can be made slightly easier by using the power of recursion. The general problem that appears at every level of the recursive solution is finding whether a specific key exists in a subarray that lies between two index positions in the original array, which must appear in ascending order. Implementing the general recursive solution therefore corresponds to writing a function

```plaintext
function binarySearch(key, array, p1, p2)
```

which returns `true` if the key appears in the subarray specified by `p1` and `p2`.

Before proceeding further, it is important to clarify exactly what the indices `p1` and `p2` signify. There are several conventions one might adopt for specifying a range of elements in an array. One could, for example, provide the starting and ending indices or, alternatively, the starting index and the number of elements. Many modern languages, including JavaScript, follow a slightly different convention that at first seems counterintuitive, but usually leads to simpler code. As you would expect, the argument `p1` refers to the index of the first element in the desired subarray. The argument `p2`, however, refers to the index just after the last element in that subarray. In other words, `p2` indicates the index of the first element that is not part of the desired subarray. Given that JavaScript uses this convention whenever it specifies the boundaries of a subarray or substring, it makes sense to adopt the same interpretation for the arguments to `binarySearch` so as to reduce the possibility of confusion.

The code for `binarySearch` itself must first check to see if the remaining subarray is empty. In that case, the key does not appear. If there are any elements, the next step is to compute the index position of the middle element of the subarray, which is the average of the first and last index position, converted to an integer, if necessary. Given that the last index position in the subarray is at position `p2 - 1`, the index in the middle of the subarray is given by the formula

\[
\left\lfloor \frac{p_1 + p_2 - 1}{2} \right\rfloor
\]

The square brackets containing horizontal marks only at the bottom indicate the mathematical function called `floor` (which is written as `Math.floor` in JavaScript), which returns the largest integer that does not exceed the argument. For example, if `p1` and `p2` have the values 0 and 50, the use of the `Math.floor` function ensures that the midpoint appears at 24 instead of 24.5, which is not a valid array index.

Once you have the index of the midpoint, the rest of the code is straightforward. If the key appears at the midpoint index, you’ve found it. If not, you simply make a recursive call on the appropriate subarray. The finished code appears in Figure 7-3.
FIGURE 7-3 Implementation of the binary search algorithm

/
* File: BinarySearch.js
* 
* This file illustrates the binary search algorithm.
*/

var STATE_CODES = [
"AK", "AL", "AR", "AZ", "CA", "CO", "CT", "DE", "FL", "GA",
"HI", "IA", "ID", "IL", "IN", "KS", "KY", "LA", "MA", "MD",
"ME", "MI", "MN", "MO", "MS", "MT", "NC", "ND", "NE", "NH",
"NJ", "NM", "NV", "NY", "OH", "OK", "OR", "PA", "RI", "SC",
"SD", "TN", "TX", "UT", "VA", "VT", "WA", "WI", "WV", "WY"
];

/
* Function: existsIn
* Usage: if (existsIn(key, array)) . . .
* 
* Returns true if the key exists in the array and false otherwise.
* In this implementation, the elements of the array must appear in
* ascending order.
*/

function existsIn(key, array) {
    return binarySearch(key, array, 0, array.length);
}

/
* Function: binarySearch
* Usage: if (existsIn(key, array, p1, p2)) . . .
* 
* Returns true if the key exists in the sorted subarray indicated by the
* array argument and the indices p1 and p2. For consistency with other
* array functions in JavaScript, the subarray begins at p1 and extends
* up to but not including p2.
*/

function binarySearch(key, array, p1, p2) {
    if (p1 >= p2) return false;
    var mid = Math.floor((p1 + p2 - 1) / 2);
    if (key === array[mid]) {
        return true;
    } else if (key < array[mid]) {
        return binarySearch(key, array, p1, mid);
    } else {
        return binarySearch(key, array, mid + 1, p2);
    }
}
7.2 Computational complexity

Intuitively, using binary search to find a key value in a sorted array seems more efficient than using linear search because you can rule out half of the remaining elements on each cycle, without ever looking at them. For computer scientists, however, it is important to be able to quantify that difference in efficiency. How could you establish that binary search is the more efficient algorithm? Perhaps more importantly, how could you express the relative efficiency of the two search techniques in a way that makes their differences easy to compare mathematically.

One possible technique for determining the efficiency of an algorithm is to run it with a variety of input values and measure explicitly how much time it takes to complete the calculation. The empirical approach, however, tells you much more about the particular coding of an algorithm and the conditions under which that algorithm is performed. For example, if I recoded a JavaScript program in a more efficient language, I would likely see a dramatic increase in speed. That difference, however, has nothing to do with the algorithm itself. What’s more, the efficiency gain that comes from translating the program into a different language applies equally to both linear and binary search and therefore provides no insight into the relative efficiency. As a general rule, it is more useful to find a qualitative measure that gives you insight into algorithmic performance than a quantitative measure that depends on so many extraneous factors.

The notion of problem size

The most valuable qualitative insights you can obtain about algorithmic efficiency are usually those that help you understand how the performance of an algorithm responds to changes in problem size. Problem size is usually easy to quantify. For algorithms that operate on numbers, it generally makes sense to let the numbers themselves represent the problem size. For most algorithms that operate on arrays or arrays, you can use the number of elements. When evaluating algorithmic efficiency, computer scientists traditionally use the letter $N$ to indicate the size of the problem, no matter how it is calculated.

The relationship between $N$ and the performance of an algorithm as $N$ becomes large is called the computational complexity of that algorithm. In general, the most important measure of performance is execution time, although it is also possible to apply complexity analysis to other concerns, such as the amount of memory space required. Unless otherwise stated, all assessments of complexity used in this book refer to execution time.
Worst-case performance of the search algorithms

In some cases, the running time of an algorithm depends not only on the size of the problem but on the specific data values you choose. If you are lucky enough to search for the first element in an array, the linear search algorithm runs very quickly and requires only a single iteration of the for loop. Similarly, if the key you’re looking for just happens to be exactly in the middle of a sorted array, the binary search algorithm finds it on the first try. These situations are unusual. If most cases, both search algorithms will require more than this minimum amount of time.

When you analyze the computational complexity of a program, you’re usually not interested in the best possible performance. It is usually more important to know how an algorithm will perform in the worst possible case, because that analysis makes it possible to set an upper bound on the computational complexity. If you analyze for the worst case, you can guarantee that the performance of the algorithm will be at least as good as your analysis indicates. You might sometimes get lucky, but you can be confident that the performance will not get any worse.

As a first step toward developing a measure of computational complexity, it is often useful to count the number of times some common operation is performed during the execution of an algorithm. For the search algorithms, for example, the operation that occurs as frequently as any other is a comparison between the key and a value in the array. For each of the two algorithms, one can therefore ask the following question:

How many comparisons are required to search for a key value in an array of length \( N \) in the worst possible case?

For the linear search algorithm, the worst case occurs when the key is in the last position or does not appear in the array at all. In either of those cases, the `existsIn` implementation will have to perform \( N \) string comparisons.

Analyzing the binary search algorithm requires a little more thought. At every level of the recursion, the code performs two string comparisons: one to test whether you’ve found the key and one to determine whether you should make the recursive call on the first or second half of the subarray. The total number of comparisons is therefore dependent on the number of recursive calls. Each call cuts the number of elements in half, and you can always stop when you get down to a single element or an empty subarray. Thus, the number of recursive calls is the number of times you can divide the original array size in half until you reach 1, as illustrated in the following calculation:

\[
1 = \frac{N}{2} / 2 / \cdots / 2 / 2
\]
Multiplying all those 2s together gives the equation

\[ 1 = N / 2^k \]

or, equivalently

\[ 2^k = N \]

If you remember logarithms from high-school algebra, you can express the value of \( k \), which corresponds to the maximum number of recursive levels, as follows:

\[ k = \log_2 N \]

Given that there are two string comparisons at every level, the worst-case number of comparisons is therefore

\[ 2 \log_2 N \]

Thus, in the worst case, finding a key in an array of \( N \) elements requires \( N \) comparisons if you use linear search and \( 2 \log_2 N \) comparisons if you use binary search. For most people, however, such formulas do not convey a real sense of how these algorithms compare. For that, you need to look at some numbers. The following table shows the closest integer to \( 2 \log_2 N \) for various values of \( N \).

<table>
<thead>
<tr>
<th>( N )</th>
<th>( 2 \log_2 N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>100</td>
<td>13</td>
</tr>
<tr>
<td>1,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000</td>
<td>27</td>
</tr>
<tr>
<td>100,000</td>
<td>33</td>
</tr>
<tr>
<td>1,000,000</td>
<td>40</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>60</td>
</tr>
</tbody>
</table>

Reflecting on what the values in this table mean, you can see that, for small arrays, both strategies work reasonably well. On the other hand, if you have an array with 1,000,000,000 elements, linear search requires 1,000,000,000 comparisons to search that array in the worst case, whereas the binary search algorithm gets the job done using at most 60 comparisons. Clearly, this reduction in the number of required comparisons represents an overwhelming increase in algorithmic efficiency.

**Big-O notation**

Computer scientists traditionally use a shorthand form called **big-O notation** to denote the computational complexity of algorithms. The German mathematician Paul Bachmann introduced big-O notation long before computers were invented. It
first appears on page 400 of his 1892 book entitled *Analytische Zahlentheorie* (*Analytic Number Theory*) in the paragraph shown in Figure 7-4. My translation of his German reads as follows:

When we express by the notation $O(n)$ a quantity whose size relative to $n$ does not exceed the order of $n$; whether it actually contains terms of order $n$ in itself remains undecided at the end of the foregoing process.

Bachmann’s intent with this notation was to establish a bound on the error of an approximation. In the case of equation 11 in his book, the expression $n \log n$ is an accurate approximation of the function $\tau(n)$ up to some error term on the order of $n$. Occasions to express precisely this form of approximate bounds come up all the time in algorithmic analysis, and Bachmann’s notation has proven to be a useful tool even in this more technological age.

As you can see from Bachmann’s example in Figure 7-4, big-O notation is very simple and consists of the letter $O$ followed by a formula enclosed in parentheses. That formula is typically a simple function involving the problem size $N$. For example, in this chapter you will encounter the big-O expression

$$O(N^2)$$

which reads aloud as “big-oh of $N$ squared.”

Big-O notation is used to specify qualitative approximations and is therefore ideal for expressing the computational complexity of an algorithm. Coming as it does from mathematics, big-O notation has a precise definition, which is beyond the scope of this book. As a general rule, it is far more important for you to understand what big-O means from a more intuitive point of view.

When you use big-O notation to estimate the computational complexity of an algorithm, the goal is to provide a qualitative insight as to how changes in $N$ affect the algorithmic performance as $N$ becomes large. Because big-O notation is not
intended to be a quantitative measure, it is not only appropriate but desirable to reduce the formula inside the parentheses so that it captures the qualitative behavior of the algorithm in the simplest possible form. The most common simplifications that you can make when using big-O notation are as follows:

1. **Eliminate any term whose contribution to the total ceases to be significant as N becomes large.** When a formula involves several terms added together, one of the terms often grows much faster than the others and ends up dominating the entire expression as N becomes large. For large values of N, this term alone will control the running time of the algorithm, and you can ignore the other terms in the formula entirely.

2. **Eliminate any constant factors.** When you calculate computational complexity, your main concern is how running time changes as a function of the problem size N. Constant factors have no effect on the overall pattern. If you bought a new machine that was twice as fast as your old one, any algorithm runs twice as fast as before for every value of N. The growth pattern, however, remains the same. Thus, you can ignore constant factors when you use big-O notation.

Big-O notation makes it possible to express the computational complexity of the linear and binary search algorithms in a concise form that follows closely from the earlier strategy of counting the number of comparisons involved. The actual running time of the program will depend on operations other than the comparisons, but those operations can subdivided into groups based on how often they are executed. In the case of linear search, for example, the running time is the sum of certain operations that are executed on each of the N cycles of the loop and some that are executed only once, such as those involved in the initialization of the for loop. The precise running time depends, of course, on the speed of the hardware, but it can be expressed in general as the sum

$$\alpha N + \beta$$

where \(\alpha\) is the time required to execute one loop cycle and \(\beta\) is the constant overhead of setting up the process.

It would, however, not be particularly useful to say that the running time of selection sort is

$$O(\alpha N + \beta)$$

even if you knew the values of the constants \(\alpha\) and \(\beta\). The entire point of using big-O notation is that doing so enables you to simplify the formula so that it provides an assessment of computational complexity that makes it easy to predict how running time grows as a function of N. The constant term \(\beta\) becomes less
significant as \( N \) becomes large and can therefore be eliminated by the first simplification rule. The second rule allows you to dispense with the constant \( \alpha \) as well, which has no effect on the comparative running time for different values of \( N \). The standard way to express the complexity of linear search is therefore simply

\[
O(N)
\]

By a similar argument, the complexity of binary search is

\[
O(\log N)
\]

The fact that there are two comparisons for each recursive level doesn’t matter, because that doubling is simply a constant factor. More subtly, it is also appropriate to eliminate the subscript 2 from the log function, because logarithms computed using different bases differ only by a constant factor. For this reason, it is traditional to omit the logarithmic base when you use big-O notation.

### 7.3 Sorting

The best way to appreciate the importance of algorithmic analysis is to consider a problem domain in which different algorithms vary widely in their performance. Of these, one of the most interesting problems is that of *sorting*, which consists of rearranging the elements in an array so that they appear in some defined order. For example, suppose you have the following values in the variable `array`, as follows:

<table>
<thead>
<tr>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>314</td>
</tr>
</tbody>
</table>

Your mission is to write a function `sort(array)` that rearranges the elements into ascending order, like this:

| 159 | 264 | 265 | 314 | 323 | 358 | 846 | 979 |

#### The selection sort algorithm

There are many algorithms you could choose to sort an array of integers into ascending order. One of the simplest is called *selection sort*. Given an array of size \( N \), the selection sort algorithm goes through each element position and finds the value that should occupy that position in the sorted array. When it finds the appropriate element, the algorithm exchanges it with the value that previously occupied the desired position to ensure that no elements are lost. Thus, on the first cycle, the algorithm finds the smallest element and swaps it with the first element, which appears at index position 0 in JavaScript. On the second cycle, it finds the
smallest remaining element and swaps it with the second element. Thereafter, the algorithm continues this strategy until all positions in the array are correctly ordered. An implementation of `sort` that uses selection sort is shown in Figure 7-5.

For example, if the initial contents of the array are

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>314</td>
<td>159</td>
<td>265</td>
<td>358</td>
<td>979</td>
<td>323</td>
<td>846</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

the first cycle through the outer `for` loop identifies the 159 in index position 1 as the smallest value in the entire array and then swaps it with the 314 in index position 0 to leave the following configuration:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>159</td>
<td>314</td>
<td>265</td>
<td>358</td>
<td>979</td>
<td>323</td>
<td>846</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

On the second cycle, the algorithm finds the smallest element between positions 1 and 7, which turns out to be the 264 in position 7. The code then exchanges these elements, ensuring that the first two positions have the correct values, as follows:
On each subsequent cycle, the algorithm performs a swap operation to move the next smallest value into its appropriate final position. When the for loop is complete, the entire array is sorted.

**Analyzing the performance of selection sort**

Although selection sort is relatively easy to understand, it turns out to be inefficient when applied to large arrays. To determine the computational complexity, it helps to think about what the algorithm has to do on each cycle of the outer loop. To correctly determine the first value in the array, the selection sort algorithm must consider all \( N \) elements as it searches for the smallest value. Thus, the time required on the first cycle of the loop is presumably proportional to \( N \). For each of the other elements in the array, the algorithm performs the same basic steps but looks at a smaller number of elements each time. It looks at \( N-1 \) elements on the second cycle, \( N-2 \) on the third, and so on, so the total running time is proportional to

\[
N + N-1 + N-2 + \ldots + 3 + 2 + 1
\]

Because it is difficult to work with an expression in this expanded form, it is useful to simplify it by applying a bit of mathematics. As you may have learned in an algebra course, the sum of the first \( N \) integers is given by the formula

\[
\frac{N \times (N + 1)}{2}
\]

or, multiplying out the numerator,

\[
\frac{N^2 + N}{2}
\]

Although it would be mathematically correct to use this formula directly in the big-O expression

\[
O\left(\frac{N^2 + N}{2}\right)
\]

you would never do so in practice because the formula inside the parentheses is not expressed in the simplest form. As \( N \) increases, \( N^2 \) quickly dominates \( N \), which allows you to eliminate the smaller term from the expression. You can, moreover, eliminate the constant factor and express the complexity of selection sort as

\[
O(N^2)
\]
This expression captures the essence of the performance of selection sort. As the size of the problem increases, the running time tends to grow by the square of that increase. Thus, if you double the size of the array, the running time goes up by a factor of four. If you instead multiply the number of input values by 10, the running time explodes by a factor of 100.

The $O(N^2)$ complexity of selection sort makes it inappropriate for use with anything other than very small arrays. The same is true for most sorting algorithms that process the elements of the array in a linear order. To develop a better sorting algorithm, you need to adopt a different approach.

**The power of divide-and-conquer strategies**

Oddly enough, the insight you need to find a better strategy is that the quadratic behavior of algorithms like selection sort has a hidden virtue. The fundamental characteristic of quadratic complexity is that, as the size of a problem doubles, the running time increases by a factor of four. The reverse, however, is also true. If you divide the size of a quadratic problem by two, you decrease the running time by that same factor of four. This fact suggests that dividing an array in half and then applying a recursive divide-and-conquer approach might reduce the required time.

To make this idea more concrete, suppose you have a large array that you need to sort. What happens if you divide the array into two halves and then use the selection sort algorithm to sort each of those pieces? Because selection sort is quadratic, each of the smaller arrays requires one quarter of the original time. You need to sort both halves, of course, but the total time required to sort the two smaller arrays is still only half the time that would have been required to sort the original array. If it turns out that sorting two halves of an array simplifies the problem of sorting the complete array, you will be able to reduce the total time substantially. More importantly, once you discover how to improve performance at one level, you can use the same algorithm recursively at ever level of the decomposition.

To determine whether a divide-and-conquer strategy is applicable to the sorting problem, you need to answer the question of whether dividing an array into two smaller arrays and then sorting each one helps to solve the general problem. As a way to gain some insight into this question, suppose that you start with the same array used in the selection sort example:

<table>
<thead>
<tr>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>314</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
If you divide the array of eight elements into two arrays of length four and then sort each of those smaller arrays, you get the following situation in which each of the smaller arrays is sorted:

<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>159</td>
<td>264</td>
</tr>
<tr>
<td>265</td>
<td>323</td>
</tr>
<tr>
<td>314</td>
<td>846</td>
</tr>
<tr>
<td>358</td>
<td>979</td>
</tr>
</tbody>
</table>

How useful is this decomposition? Remember that the goal is to take the values out of these smaller arrays and put them back into the original array in the correct order. How does having these smaller sorted arrays help you accomplish that goal?

**Merging two arrays**

As it happens, reconstructing the complete array from the smaller sorted arrays is a much simpler problem than sorting itself. The required technique, called *merging*, depends on the fact that the first element in the complete ordering must be either the first element in \( a_1 \) or the first element in \( a_2 \), whichever is smaller. In this example, the first element you want in the array is the 159 in \( a_1 \). If you store that element back in the first position of \( a_1 \) and then remove it from further consideration, you get the following configuration:

Once again, the next element can only be the first unused element in one of the two smaller arrays. This time, you compare the 265 from \( a_1 \) against the 264 in \( a_2 \) and choose the latter:
You can easily continue this process until you have reconstructed the entire array, always choosing the smallest unused value from a1 or a2.

**The merge sort algorithm**

The merge operation, combined with recursive decomposition, gives rise to a sorting algorithm called *merge sort*, which turns out to be much more efficient than selection sort. The basic outline of the merge sort algorithm looks like this:

1. Check to see if the array is empty or has only one element. If so, it must already be sorted. This condition defines the simple case for the recursion.
2. Divide the array into two smaller arrays, each of which is half the size.
3. Sort each of the smaller arrays recursively.
4. Merge the two sorted arrays back into the original one.

The code for the merge sort algorithm, shown in Figure 7-6, divides neatly into two functions: *sort* and *merge*. The code for *sort* follows directly from the outline of the algorithm. After checking for the special case, the algorithm uses JavaScript’s *slice* method to split the array into two smaller ones, a1 and a2. The code sorts these arrays recursively and then calls *merge* to produce the solution.

Most of the work is done by the *merge* function, which takes the destination array, along with the smaller arrays a1 and a2. The indices i1 and i2 mark the progress through each of the subsidiary arrays. On each cycle of the loop, the function selects an element from a1 or a2—whichever is smaller—and adds that value to the next position in the array. As soon as the elements in either of the two smaller arrays are exhausted, the function can simply copy the elements from the other array without bothering to test them. In fact, because one of these arrays is already exhausted when the first while loop exits, the function can simply copy the rest of each array to the destination. One of these arrays will be empty, and the corresponding while loop will therefore not be executed at all.

**The computational complexity of merge sort**

You now have an implementation of the *sort* function based on the strategy of divide-and-conquer, but you do not yet have any sense of its computational complexity. When you call the merge sort implementation of *sort* on a list of N numbers, the running time can be divided into two components:

1. The amount of time required to execute the operations at the current level of the recursive decomposition
2. The time required to execute the recursive calls
/*
 * File:  MergeSort.js
 * ---------------
 * This file implements the merge sort algorithm in Javascript. The
 * steps in the merge sort algorithm are as follows:
 * 1. Split the array into two halves.
 * 2. Sort each of these smaller arrays recursively.
 * 3. Merge the two arrays back into the original one.
 */

function sort(array) {
    var n = array.length;
    if (n <= 1) return;
    var mid = Math.floor(n / 2);
    var a1 = array.slice(0, mid);
    var a2 = array.slice(mid, n);
    sort(a1);
    sort(a2);
    merge(array, a1, a2);
}

/*
 * Implementation notes: merge
 * ---------------------------
 * This function merges two sorted arrays, a1 and a2, into the original
 * array, which is passed as the first argument. The contents of array
 * are overwritten, but it should have the same number of elements as in
 * a1 and a2 combined. Given that the arrays a1 and a2 are sorted, the
 * implementation can always select the first unused element in one of
 * these arrays to fill the next position in the original array. The
 * index variables i, i1, and i2 keep track of the current positions in
 * array, a1, and a2, respectively.
 */

function merge(array, a1, a2) {
    var n1 = a1.length;
    var n2 = a2.length;
    var i = 0;
    var i1 = 0;
    var i2 = 0;
    while (i1 < n1 && i2 < n2) {
        if (a1[i1] < a2[i2]) {
            array[i++] = a1[i1++];
        } else {
            array[i++] = a2[i2++];
        }
    }
    while (i1 < n1) array[i++] = a1[i1++];
    while (i2 < n2) array[i++] = a2[i2++];
}
At the top level of the recursive decomposition, the cost of performing the nonrecursive operations is proportional to $N$. The `slice` method copies the subarrays, which requires time proportional to the array size, and the call to `merge` has the effect of refilling the original $N$ positions in the array. If you add these operations and ignore the constant factor, you discover that the complexity of any single call to `sort`—not counting the recursive calls within it—is $O(N)$.

But how would you determine the cost of the recursive operations? The easiest way to get a sense for how these costs accumulate is to look at a picture of the recursive decomposition, as shown in Figure 7-7. As you move down through the recursive hierarchy, the arrays get smaller, but more numerous. The amount of work done at each level is always directly proportional to $N$. Figuring out the number of recursive levels follows the same logic as in the analysis of binary search. The number of recursive levels is the number of times you can divide $N$ by 2 until you reach the value 1, which is $\log_2 N$. Because the number of levels is $\log_2 N$ and the amount of work done at each level is proportional to $N$, the overall complexity is proportional to the product of these values and is therefore $O(N \log N)$.

**Comparing $N^2$ and $N \log N$ performance**

But how much better is an algorithm that runs in $O(N \log N)$ time than one that requires $O(N^2)$? One way to assess the level of improvement is to look at empirical data to get a sense of how the running times of the selection and merge sort algorithms compare. On my laptop, the actual running times for the two sorting strategies look like this:

---

**Figure 7-7** Recursive decomposition of merge sort

*Sorting a vector of size $N*

- Requires sorting two vectors of size $N/2$:
  - 2 $\times$ $N/2$ operations

- Requires sorting four vectors of size $N/4$:
  - 4 $\times$ $N/4$ operations

- Requires sorting eight vectors of size $N/8$:
  - 8 $\times$ $N/8$ operations

*and so on.*
### 7.4 Standard complexity classes

In programming, most algorithms fall into one of several common complexity classes, several of which are shown in Figure 7-8. The differences in efficiency

<table>
<thead>
<tr>
<th>$N$</th>
<th>Selection sort</th>
<th>Merge sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0000024 sec</td>
<td>0.000128 sec</td>
</tr>
<tr>
<td>100</td>
<td>0.000169 sec</td>
<td>0.000196 sec</td>
</tr>
<tr>
<td>1,000</td>
<td>0.0159 sec</td>
<td>0.00236 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>1.58 sec</td>
<td>0.027 sec</td>
</tr>
<tr>
<td>100,000</td>
<td>158.7 sec</td>
<td>0.324 sec</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15,780.0 sec (4:23:32)</td>
<td>3.892 sec</td>
</tr>
</tbody>
</table>

For 10 items, this implementation of merge sort is more than five times slower than selection sort. At 100 items, selection sort is still faster, but not by very much. By the time you get up to 100,000 items, merge sort is almost 500 times faster than selection sort. With 1,000,000 items, selection sort takes over four hours to finish while merge sort completes the job in less than four seconds.

You can get much the same information by comparing the computational complexity formulas for the two algorithms, as follows:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N^2$</th>
<th>$N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>664</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>9,966</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>132,877</td>
</tr>
<tr>
<td>100,000</td>
<td>10,000,000,000</td>
<td>1,660,964</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000,000,000</td>
<td>19,931,569</td>
</tr>
</tbody>
</table>

The numbers in both columns grow as $N$ becomes larger, but the $N^2$ column grows much faster than the $N \log N$ column. Sorting algorithms based on an $N \log N$ algorithm will therefore be useful over a much larger range of array sizes.

#### Figure 7-8

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Big O Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
<td>Returning the first element in a vector</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log N)$</td>
<td>Binary search in a sorted vector</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(N)$</td>
<td>Linear search in a vector</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>$O(N \log N)$</td>
<td>Merge sort</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(N^2)$</td>
<td>Selection sort</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(2^N)$</td>
<td>Tower of Hanoi puzzle</td>
</tr>
</tbody>
</table>
between these classes are in fact profound. You can begin to get a sense of how the complexity classes stand in relation to one another by looking at the graph in Figure 7-9, which plots these complexity functions on a traditional linear scale.

Unfortunately, this graph tells an incomplete and somewhat misleading part of the story, because the values of \( N \) are all very small. Complexity analysis, after all, is primarily relevant as the values of \( N \) become large. Figure 7-10 shows the same data plotted on a logarithmic scale, which gives you a better sense of how these functions grow over a more extensive range of values.

Algorithms that fall into the constant, linear, and quadratic complexity classes are all part of a more general family called \textit{polynomial algorithms} that execute in time \( N^k \) for some constant \( k \). One of the useful properties of the logarithmic plot shown in Figure 7-10 is that the graph of any function \( N^k \) always comes out as a straight line whose slope is proportional to \( k \). If you look at the figure, it is clear that the function \( N^k \)—no matter how big \( k \) happens to be—will invariably grow more slowly than the exponential function represented by \( 2^N \), which continues to curve upward as the value of \( N \) increases. This property has important implications in terms of finding practical algorithms for real-world problems. Even though the
selection sort example demonstrates that quadratic algorithms have substantial performance problems for large values of $N$, algorithms whose complexity is $O(2^N)$ are considerably less efficient.

By convention, computer scientists designate problems that can be solved using algorithms that run in polynomial time as *tractable*, in the sense that they are amenable to implementation on a computer. Problems for which no polynomial-time algorithm exists are regarded as *intractable*.

Unfortunately, there are many commercially important problems for which all known algorithms require exponential time. Many of these problems fall into the class of *NP-complete problems*, which are described in more detail in Chapter 10. As far as anyone knows, it is not possible to solve any of the NP-complete problems in polynomial time. At least for the moment, the optimal solution is to try every possibility in an ever-expanding decision tree, which requires exponential time. On the other hand, no one has been able to prove conclusively that no polynomial-time algorithm for these problems exist. There might be some clever algorithm that would make NP-complete problems tractable. If so, many problems currently believed to be difficult would become significantly easier to solve.
At the beginning of the 20th century, mathematicians became interested in the possibility of finding algorithmic techniques for proving mathematical theorems that could be carried out mechanically. In an address before the International Congress of Mathematicians in 1928, the eminent German mathematician David Hilbert formulated what he called the Entscheidungsproblem, for which the clearest English translation is decision problem. The essence of the Entscheidungsproblem can be expressed in the following question:

Is it possible to find a mechanical procedure that can determine, given a specific proposition in a formal system of symbolic logic, whether that proposition is provable within that system?

A few years later, a 23-year-old graduate student at Cambridge named Alan Turing became intrigued by Hilbert’s problem. After working on the problem for several months, Turing was able to show that Hilbert’s dream of finding such an algorithmic process was to remain forever unfilled. In his 1936 paper entitled “On computable numbers, with an application to the Entscheidungsproblem,” Turing was able to show that there can never be a mechanical procedure that will always be able to decide whether a proposition is provable. The details of Turing’s proof are beyond the scope of this book, but you will have a chance to learn some of the central ideas in the discussion of uncomputable functions in Chapter 9.

Today, Turing’s paper is remembered less for its mathematical contribution to the Entscheidungsproblem—which had actually been settled earlier in 1936 by Alonzo Church at Princeton—than it is for the abstract model of computation Turing developed to carry out his proof. That model, which has ever since been
The life of Alan Turing

As you will discover in reading this book, Alan Turing is one of the giants of computer science. His work in computability described in this chapter lies at the foundation of theoretical computer science and remains directly relevant today. In addition, Turing made critical contributions in several other areas, including hardware design, cryptography, artificial intelligence, and philosophy. Beyond his technical contributions, however, Turing is also interesting because his life and its tragic end tell a compelling story about the tensions between the individual and society.

Alan Mathison Turing was born in London in 1912. Turing’s parents, Julius and Ethel Turing, spent the years of Alan’s childhood in India, where Julius served as a colonial administrator in Madras. Alan and his brother John remained in England, where they were brought up in foster homes. After attending boarding school, Turing entered King’s College at Cambridge in 1931, where he earned a distinguished degree in mathematics in 1934, spending his free time as a rower and long-distance runner. Over the next two years, Turing remained at Cambridge, where he was introduced to Hilbert’s Entscheidungsproblem, described at the beginning of this chapter. His paper on computable numbers, which introduced the ideas that underlie the theory of computation, was published in 1936, when Alan Turing was just 24 years old.

With the beginning of the World War II, Turing turned his attention to cryptography. As you will discover in Chapter 11, Turing and his colleagues were able to crack the German Enigma code. In 1946, Turing received the OBE, Britain’s highest honor, for his cryptography work during the war.

In later years, Turing continued to have an amazingly productive intellectual career, designing one of the earliest digital computers, exploring the potential of computational biology, and laying out the philosophical foundations for artificial intelligence.

It was during this time that another aspect of Turing’s character was to have a profound effect on his life. Ever since childhood, when he developed a strong attraction for Christopher Morcom, an older student at Turing’s boarding school who died of tuberculosis in 1930, Turing recognized himself as a homosexual. Having never gone out of his way to hide his sexuality, Turing was arrested in 1952 on a charge of “gross indecency”—the same Victorian-era statute used against Oscar Wilde half a century before.

After his arrest, Turing’s world began to come apart. Now regarding Turing as a security risk, the British secret service imposed restrictions on Turing’s ability to work in cryptography and to interact with foreign mathematicians. Turing found these constraints difficult to bear, particularly given his deeply held conviction that he had done nothing wrong. In his play Breaking the Code, English playwright Hugh Whitemore has Turing express his frustration to a British officer as follows:

It took more than mathematics and electronic ingenuity to crack the U-boat enigma. It required determination, tenacity, moral fibre, . . . my love of my country. You trusted me then, why not now?

This loss of trust proved intolerable to Turing, who committed suicide on June 8, 1954 by eating a cyanide-poisoned apple.
known as a \textit{Turing machine}, has played an essential role in the development of computer science and serves as a foundation for the theory of computability.

\section*{8.1 Turing’s model of computation}

Because Turing’s paper on the \textit{Entscheidungsproblem} appeared in a mathematical journal, it is hardly surprising that parts of it are difficult to follow without advanced mathematical training. Several parts of the paper, however, are extremely easy to read. In the section reprinted as Figure 8-1, Turing offers a general description of the computing process by appealing to the reader’s intuition as to how one does computation by hand. In that description, Turing uses the word \textit{computer} to refer to a human being engaged in computation; humans, after all, were the only computers around in Turing’s day.

One of the central purposes of Turing’s intuitive description of computing is to identify those aspects of the process that are fundamental, which in turn makes it possible to simplify computing to its essential elements. In doing so, Turing developed a model in which all computation is carried out on a one-dimensional tape divided into squares, like this:

\begin{verbatim}
. . .   . . .   . . .   . . .   . . .   . . .   . . .   . . .   . . .   . . .   . . .
\end{verbatim}

The dots at each end of the diagram indicate that the tape extends arbitrarily far in each direction. It is not necessary that the tape be \textit{infinite}, which would of course make it impossible to build a physical realization of the machine. It is sufficient for the tape to be \textit{unbounded}. If, in the process of carrying out some computation, you discover that the tape is running out, it must always be possible to splice on additional tape. Intuitively, as problems become more complex, the Turing machine used to solve them will require more computational space, which is precisely what the squares on the tape represent. If you want to assess—as Turing did—the fundamental limits of computing, you can’t allow yourself to be constrained by any arbitrary physical limitations. In particular, you should never conclude that a problem is unsolvable simply because you run out of scratch paper or, in the Turing machine model, the machine runs out of tape.

Each square on the tape holds a single symbol chosen from a finite collection, which is called the \textit{alphabet} of the machine. In his informal description, Turing argues that restricting the alphabet to a finite set of symbols does not fundamentally limit the computational power of his machine, noting that “it is always possible to use sequences of symbols in the place of single symbols.” The same argument suffices to reduce the number of symbols to two. As long as there are two symbols in your alphabet—using only one would mean that every square on the tape would
Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child’s arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on a one-dimensional paper, i.e., on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. The effect of this restriction on the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. . . . The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 999999999999999 and 999999999999999 are the same.

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his “state of mind” at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be “arbitrarily close” and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

Let us imagine the operations performed by the computer to be split up into “simple operations” which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always “observed” squares.

Besides these changes of symbols, the simple operations must include changes of distributions of observed squares. The new observed squares must be immediately recognizable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square. . . .

The simple operations must therefore include:

(a) Changes of the symbol on one of the observed squares.

(b) Changes of one of the squares observed to another square within L squares of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

(A) A possible change (a) of symbol together with a possible change of state of mind.

(B) A possible change (b) of observed squares, together with a possible change of state of mind.

The operation actually performed is determined, as has been suggested . . . by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an “m-configuration” of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned squares. The move which is done, and the succeeding configuration, are determined by the scanned symbol and the m-configuration. The machines as defined here can be constructed to compute the same sequence computed by the computer.
of necessity contain the same symbol—you can encode other symbols in precisely
the same way that modern computers encode characters as a sequence of bits. In
another section of his paper, Turing himself suggests using the binary digits 0 and 1
as the symbols for the machine. Today, the idea that computers should use binary
representation for their internal arithmetic is firmly established, although it was a
subject of intense controversy during the 1940s, when most of the early computers
relied on decimal arithmetic. In a sense, Turing’s paper anticipated—and may
perhaps have guided—the subsequent evolution of computer architecture toward
binary arithmetic.

The Turing machines described in this book use 0 and 1 as their alphabet. The
symbol 0, moreover, is used to represent blank tape. The symbol 1 appears only as
part of the input to the machine or the results of computation. Every square that is
not specifically set to contain a 1 is assumed to contain a 0 by default. Thus, a
completely blank tape is represented by a sequence of 0 symbols, extending
arbitrarily far in each direction.

Reasoning from his analogy to the processes undertaken by human computers,
Turing concludes that computation need not focus on the entire tape at once. Just as
a person computing on paper looks at only part of the computation at any one time,
a Turing machine should be able to limit its focus to a finite set of squares; later in
his paper, Turing argues that a single square will suffice. Any computation that
could be carried out by looking at a larger set of squares can also be done—with a
little more work—by limiting one’s attention to a single square at a time. Thus, the
conceptual model of the Turing machine includes the notion of a tape head, which
focuses on a particular square. In the diagrams that follow, the tape head is
indicated by a bold square outlining one of the squares on the tape, like this:

```
... 0 0 0 0 0 [ ] 0 0 0 0 0 ...
```

All activity of the Turing machine is limited to the symbol marked by the current
position of the tape head, which is called the current symbol. In making decisions
as to which operations to perform, a Turing machine program can look only at the
current symbol and not at the contents of any other tape square. Turing machine
programs, however, can change the symbol in the current square and thereby change
the configuration on the tape. Moreover, each instruction in a Turing machine
program moves the tape head left or right one position so that it scans a new square.
Thus, the focus of a Turing machine shifts as the computation proceeds.

The one remaining concept expressed in Turing’s description of his idealized
computational model is the notion of a “state of mind.” Here, Turing expresses the
idea that the behavior of his machine depends not only on the current symbol, but
also on some internalized computational state maintained by the computer, whether that computer is a human or a machine.

A simple way to understand this concept of a state of mind is to imagine that the instructions for the machine are printed in a book. Each page in the book contains a pair of instructions: one to be executed if the current symbol is a 0 and another to be executed if that symbol is a 1. The instructions specify whether the machine should write a new symbol on the tape and which direction the tape head should move; in addition, however, they specify a page number. Some instructions may specify staying on the same page, in which case the machine will execute the same set of instructions in the new configuration. Others will specify a different page number that contains other instructions. The current page number represents the current state of the machine.

Thus, each instruction in a Turing machine program must specify three things:

1. The new symbol to be written to the tape, overwriting the current symbol
2. The direction—left or right—in which the tape head should move
3. The new current state for the machine after executing this instruction

### 8.2 A simple Turing machine

So far, the description of the Turing machine—while it follows closely from Turing's own discussion in his 1936 paper—remains relatively abstract. Like most concepts in computer science, Turing machines make the most sense after you've had a chance to play with them, and particularly when you start to write programs of your own. It is therefore useful at this point to introduce a simple Turing machine program.

Although there are many ways to describe Turing machine programs, I find that they are easiest to follow when presented in a tabular form, like this:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1R2</td>
<td>1R0</td>
</tr>
<tr>
<td>2</td>
<td>1L2</td>
<td>1L1</td>
</tr>
</tbody>
</table>

In these examples, the numbers running down the left-hand side represent the state numbers in the program. This Turing machine program, for example, has two states and therefore corresponds in the instruction-book metaphor to a two-page book of instructions. The 0 and 1 at the top of the columns indicate the two possible symbols one might find in the current tape square. Whenever the Turing machine executes an instruction, it finds the entry in the row and column of the table corresponding to the current state and symbol and executes the instruction it finds
there. Each instruction is a string consisting of three parts: the new symbol to be written on the tape (0 or 1), the direction in which the tape head should move (L or R), and the new state number. The Turing machine always begins execution in state 1 and continues executing instructions until it reaches state 0, which is used as the halting state for the machine.

What happens if you execute this Turing machine program on a blank input tape? The initial configuration of the tape looks like this:

```
0 0 0 0 0 0 0 0 0 0
```

Because the machine begins in state 1 and the current symbol is 0, the first instruction executed is the one that appears at the intersection of the row for state 1 and the column for the 0 symbol, as shown:

```
0 1
1R2 1R0
2 1L2 1L1
```

The instruction 1R2 in that box indicates that the Turing machine should write a 1 on this square, move right, and enter state 2, leaving the tape in the following configuration:

```
0 0 0 0 1 0 0 0 0 0
```

At this point, the current symbol is once again 0, but the Turing machine is now in state 2. Thus, the instruction that gets executed is now the 1L2 in the entry for state 1 and symbol 0. This instruction, writes a 1, moves right, and remains in state 2, leaving the tape looking like this:

```
0 0 0 0 1 1 0 0 0 0
```

The Turing machine is now in state 2, scanning the symbol 1. The instruction for this entry is 1L1. Since there is already a 1 on the current tape square, this instruction doesn’t need to change the contents of the tape; it simply moves left and reenters state 1 with the following tape configuration:

```
0 0 0 0 1 1 0 0 0 0
```

The machine is now in state 1 scanning a 0—just as it was at the beginning of its operation. Thus, the machine again executes the 1R2 instruction, leaving the machine in state 2 and the tape looking like this:
In this configuration, the Turing machine executes the 1L1 instruction, so that the machine returns to state 1, with the tape looking like this:

\[ \cdots 0 0 0 0 1 1 1 0 0 0 \cdots \]

The Turing machine is still going. The active instruction in the current state is 1R0, which causes the machine to keep the 1 on the current tape square, move right, and halt its execution. The final state of the tape is therefore

\[ \cdots 0 0 0 0 1 1 1 0 0 0 \cdots \]

To understand how Turing machines work, it helps enormously to trace through their operation by hand. Using pencil and paper, simulate the execution of the following Turing machine programs, starting with the input tape shown:

**8.3 Computing mathematical functions**

Traditionally, computation has been concerned primarily with arithmetic and various higher-level operations from mathematics. If a Turing machine is to offer a useful model of computation, it must prove its worth in the mathematical domain. Thus, it must be possible, for example, to implement simple mathematical operations like addition as Turing machine programs.
To simplify the representation, the Turing machine programs in this book limit
their attention to the natural numbers, which consist of the nonnegative integers
0, 1, 2, 3, and so on. Although it is tempting to represent such numbers in binary
notation using zeros and ones, that approach doesn’t really work. Given that 0 and
1 can each appear in binary notation, there is no way to tell where a particular
number starts and stops. With only two symbols, the usual approach is to represent
numbers in unary notation, in which a particular number is recorded as a sequence
of that many 1s. In unary notation, the symbol 0 can serve as a numeric separator,
making it much easier to record a series of numbers on the Turing machine tape.
For example, if you wanted to prepare an input tape containing the numbers 5 and 3,
you would do so as follows:

\[ \cdots 0 1 1 1 1 0 1 1 0 \cdots \]

By convention, numeric input tapes are positioned with the tape head scanning
the leftmost 1 in the first number, as illustrated by the preceding example.
Moreover, if more than one input value appears on the tape, each value will be
separated from the next one by a single 0 symbol.

Given this representation for input values, how would you write a Turing
machine program that adds two numbers recorded on the tape? Intuitively, all you
need to do to add two numbers is pick up the 1 at the beginning of the number and
carry it over to the 0 that separates the two values. The following Turing machine
program implements most of the necessary solution:

```
0 1
1 0R2
2 1R0 1R2
```

The bug symbol attached to this program is there to warn you that the program is
not correct as written, even though it seems to work on most examples. If you run it
starting with the tape containing the representation of the numbers 5 and 3, the final
configuration of the tape is

\[ \cdots 0 0 1 1 1 1 1 1 0 \cdots \]

which contains the desired eight 1s. The program, however, fails if either of the
numbers is 0, which is represented by a string containing no 1s at all.

This implementation of the addition program, however, has a more subtle
problem. When you add the numbers 5 and 3, as in this example, what you would
like to see as the output is the number 8. The final configuration of the tape shown
in the preceding example has the correct number of 1s, but does not display the
value 8 in its canonical form. If you prepare an input tape representing the number 8, the tape head would be over the leftmost 1 and not at some seemingly random place in the middle of the number. To make it possible to string programs together, as discussed in the section on “Functions and composition” later in this chapter, Turing machine programs that represent mathematical functions move the tape head back to the beginning of the result so that the configuration on the tape at the end of execution.

The Turing machine $M_{\text{add}}$ that appears in the margin includes instructions to move the tape head back to the beginning of the output value. It also produces the correct result if either or both of the input values are zero.

**Nonterminating machines**

Turing machine programs can be buggy in a more dramatic way. Consider, for example, the following program:

What happens if you execute this program, starting with blank tape, for example? The answer is that the machine just keeps moving right, writing down 1s forever on the unbounded tape. Such a program is an example of a nonterminating Turing machine. The following machine is also nonterminating, even though it never moves more than one square from its initial position:

No matter what symbols are on the input tape, this machine simply oscillates back and forth between the initial square and its neighbor to the right—stuck in an infinite loop.

The fact that Turing machines can fail to halt for some or all inputs is an important property of their design. Although this property might initially seem like a flaw, it turns out that any machine powerful enough to serve as a general model of computation will have this property. Nonterminating Turing machines also play a central role in establishing the limits of computability, as discussed in Chapter 9.

**The doubler function**

Turing machine programs can perform operations that are more complex than addition. The $M_{\text{doubler}}$ machine at the top of the next page, for example, doubles the value shown on the input tape, which must consist of a single unary number. Even
though this machine has only six states, it is hard to get an idea of how it works without going through the program in some detail. At the highest level, the program uses the following overall strategy:

Repeatedly erase a 1 from the original number and then write two 1s at the right end of the tape. When you have erased all of the original 1s, the new sequence will contain twice that number.

A more detailed description of the strategy, expressed in English but following the structure of the Turing machine, looks like this:

1. If you’ve reached the end of the input number, move right—which will position the tape head over the first digit of the result—and halt. If more 1s remain in the input, erase the first, start moving right, and go on to step 2.

2. Move to the right, skipping any 1s that remain in the input value. When you reach the 0 that marks the end of the number, move to the right, and proceed to step 3.

3. In this step, the goal is to find the right end of the result, which is being built up in stages as the program processes each 1 in the initial input. The first time around, the result will start immediately after the 0 separator. On subsequent iterations, the program will need to skip any 1s it has already written. In any case, this step moves to the right, skipping over 1s, until it finds the end of the result. At that point, it writes a new 1 to the result, moves right, and goes on to step 4.

4. For each 1 in the initial input, the doubler machine must write two 1s to the result area. The first is written in step 3. Step 4 writes the second of these 1s, starts moving back left, and proceeds to step 5.

5. This step moves left over any 1s that have been written to the result area. When it reaches the 0 separating the input and result areas, it skips over that as well and goes on to step 6.

6. This step continues moving left, this time skipping over the 1s in the initial input. When it reaches the left end of the input, the program turns around, moves right, and goes back to step 1, where the process starts all over again.

A complete trace of the execution of $M_{2x}$ on the input value 3 is shown in Figure 8-2. Note that the program ends its execution with the tape head positioned over the first digit of the result, thereby following the convention for mathematical functions expressed in Turing machine form.
Figure 8-2: Steps in the invocation of $M_{3,8}$ on the integer 3

The figure shows a sequence of steps involving the bitwise OR and AND operations, with each step represented by a sequence of bits.

- **OR2:** 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
- **1R2:** 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
- **1R2:** 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
- **0R3:** 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
- **1R4:** 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0R1:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1R4:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0R2:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1R5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0L7:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0R1:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1R5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0R2:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1R3:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1R3:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1R3:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1R4:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L6:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0R1:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **0R0:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
- **1L5:** 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
Functions and composition

A Turing machine program like $M_{2x}$ that takes a single numeric input value and ends its execution at the left edge of a single numeric result acts like a function in a programming language that takes one argument and returns a result. The $M_{2x}$ machine, for example, computes the function

$$f(x) = 2x$$

Many Turing machine programs other than $M_{2x}$ represent functions as well. The $M_{+3}$ program on the right, for example, adds 3 to its argument. In mathematical notation, this machine computes the function

$$g(x) = x + 3$$

The principal advantage of the convention that Turing machine functions end their execution over the leftmost 1 in a number is that Turing machine programs can then be composed in much the same way that functions are in mathematics. Suppose, for example, that you wanted to compute the function

$$h(x) = g(f(x)) = 2x + 3$$

To do so, you first have to compute the function $f$ followed by the function $g$. You can accomplish this task using a Turing machine by first executing all the steps for $M_{2x}$ and then—instead of halting—going on to execute the steps required to compute $M_{+3}$. Putting the machines together is a straightforward process. You simply replace the transitions to state 0 in the first machine with a transition to a new state beyond the ones required for that machine. You then renumber the states from the second machine so that they begin after the states you already have from the first machine. Thus, the $M_{2x+3}$ machine on the right computes the function $2x + 3$.

You could similarly construct a machine to multiply a number by 4 simply by appending two copies of the $M_{2x}$ machine. Although you can certainly accomplish the same result using a machine with fewer states, the ability to compose Turing machine functions makes it easy to construct more complicated functions from simpler building blocks, in much the same way that functions in a modern programming language allow you to decompose programs into simple components.

8.4 The Church-Turing thesis

Given how difficult they are to generate, you may be wondering why anyone would bother writing programs for a Turing machine. Because they can perform only extremely primitive computational steps, Turing machines are difficult to program
even for the simplest of operations. If you want to multiply two numbers in a
programming language like JavaScript, all you have to do is write an expression like \( x \times y \). In the context of a Turing machine, solving this problem requires a complex
program involving a large number of states. Nonetheless, it is possible to perform
multiplication on a Turing machine, even if the process seems rather tedious.

Given that a Turing machine is, after all, an extraordinarily simple computational
model, it makes sense to wonder whether or not a more sophisticated computing
paradigm, such as programs written in JavaScript, might be able to perform
computations that are beyond the reach of a Turing machine. Although no one is
certain of the answer to this question, it is not hard to establish that Turing machine
programs can do anything you could do in JavaScript or any other programming
language. Most computer scientists believe the following conjecture, which is
known as the Church-Turing thesis after Alan Turing and the Princeton-based
mathematician Alonzo Church:

No method of computation carried out by a mechanical process can be
more powerful than a Turing machine.

The word *powerful* in this sentence needs to be viewed in a relatively precise way.
Other systems of computation can certainly work faster or more efficiently. The
claim here is simply that no mechanical form of computation can solve a problem
that would not be solvable by a Turing machine, given sufficient time and space.

The Church-Turing thesis is a deeply philosophical idea, and it is not easy to
know exactly what sort of reasoning would constitute a proof of this conjecture. In
an informal sense, the most convincing argument in support of the thesis is simply
the fact that no more powerful method of computation has ever been found.
Moreover, whenever new general methods of computation are proposed, they turn
out to be precisely equivalent to—and indeed can be simulated by—a Turing
machine.

The idea of simulating one computational model in the context of another is an
important idea in the theory of computation. In Turing’s paper on computable
numbers, he relied heavily on the fact that his machine can simulate its own
operation. By adopting a suitable encoding scheme, one can take any Turing
machine program and represent it as an input tape to another Turing machine that
carries out the operation of the first. A Turing machine that can read and execute
descriptions of other Turing machine programs is called a *universal Turing
machine*. The details of such machines are beyond the scope of this book, but it is
nonetheless interesting to contemplate the implications of such a machine. The
ability to simulate an arbitrary computation within the framework of that model
turns out to be an essential characteristic of computation that stands at the
foundation of computer science theory.
Alan Turing’s greatest contribution to the intellectual history of computer science was not that he developed the machine that now bears his name, but rather that he used his vision of a general computing machine to establish a profound theoretical result about the limits of algorithmic computation. On his way to resolving Hilbert’s Entscheidungsproblem, Turing demonstrated the existence of a class of problems that are fundamentally unsolvable, in the sense that no algorithm for their solution can ever be found.

To get a sense of how a problem might indeed be unsolvable, think for a moment about the classic paradox of the irresistible force and the immovable object. Is it possible to create both a force that can move any object and an object that can never be moved? Surely that problem is unsolvable, because its very formulation creates a logical paradox. Turing discovered that much the same sort of paradox arises in computing. It is possible to describe computational problems that seem reasonable on the surface, but for which the existence of any solution would generate a contradiction.

The purpose of this chapter is to give you a sense of what such unsolvable problems look like. The examples that Turing uses in his paper on computable numbers, however, are too intricate to serve as an ideal illustration. To introduce the idea of uncomputability, this chapter begins with a more recent problem developed by Professor Tibor Radó of Ohio State University in the early 1960s. That problem—which is a wonderful mathematical puzzle in its own right—also makes it easier to demonstrate the fundamental notions of computability in a context that is considerably easier to understand.
The ideas in this chapter have connections to many disciplines besides computer science. In 1979, Douglas Hofstadter, professor of computer science at the University of Indiana, wrote a remarkable book entitled *Gödel, Escher, Bach: An Eternal Golden Braid*, which won the Pulitzer prize later that year. In a complex tapestry of interwoven narratives, dialogues, vignettes, puzzles, and literary references, the book explores how the theme of circular self-reference—an idea that Hofstadter calls “strange loops”—shows up in art and music as well as mathematics. The Dutch artist Maurits C. Escher (1898-1972), for example, created an endless staircase that seems to climb and descend at the same time. The German composer Johann Sebastian Bach (1685-1750) composed a canon in which the second voice is simply the first played backward, ending up where the original began. And Kurt Gödel (1906-1978), a Czech mathematician who came to the United States in 1939, used the same “strange loops” to discover paradoxes in the theory of mathematics. These same paradoxes have direct application to computer science and form the foundation for the theory of computability.
9.1 The Busy Beaver problem

The first example in Chapter 8 was the two-state Turing machine

```
1
0 1
1R2 1R0
1L2 1L1
```

which produces a series of three 1s when executed on blank tape. An interesting question to ask is whether any two-state machine can generate more than three 1s.

As stated, the answer to this question is certainly yes. The following machine, for example, generates a vastly larger number of 1s:

```
0
1 1
1L2 1R1
```

As you can easily verify by tracing through its operation, this machine begins by writing a 1 and then moves back and forth on the tape, continually writing new 1s at each end of an ever expanding sequence. This machine is caught in an infinite loop, writing an unbounded number of 1s. Generating an unbounded sequence of 1s turns out to be quite easy. The interesting question is therefore to determine how many 1s a machine can generate and still halt.

If you play around with two-state machines for a while, you will discover that the machine $M_{BB2}$ shown on the right generates four 1s before halting, which turns out to be the best you can do.

What about machines with three, four, or more states? How many 1s can those machines generate? This more general version of the problem, in which the question is posed as a function of the number of machine states, is called the Busy Beaver problem. The problem is most conveniently expressed in the form of a function $BB(k)$, which is defined as follows:

$$BB(k)$$ is the maximum number of 1s that can be written by a $k$-state Turing machine that halts when executed on blank tape.

Note that there is no requirement here that the 1s be consecutive or that the Turing machine should leave the tape head in any particular place. The only requirement on the Turing machine is that it must halt and not simply run on forever.
The Busy Beaver problem for small machines

For the first few values of \( k \), the Busy Beaver problem is easy to solve. Consider, for example, how you might go about finding the value of \( BB(1) \), which is the solution to the Busy Beaver problem for Turing machines with only one state. One way to solve this problem is to try all possible one-state Turing machines and see how many 1s they generate. Figure 9-1 shows each of the 64 one-state Turing machines and what output is generated when that machine is executed on blank tape. As you can see, the best you can do is write a single 1 and halt.

With two states, you have much more flexibility. As noted earlier in the chapter, having two states allows you to generate four 1s, which means that \( BB(2) \) is 4. For larger values of \( k \), however, the problem becomes more challenging. Even in the case of three-state machines, Rado reported that “there is no evidence that any known approach will yield the answer, even if we avail ourselves of high-speed computers and elaborate programs.” Three years later, Rado was able to confirm that \( BB(3) \) is 6. The value of \( BB(4) \) remained unknown until 1981, when Allen Brady at the University of Nevada was able to establish that its value is 13.

The Busy Beaver problem remains unsolved for machines with more than four states. The best-known candidate for the five-state busy beaver machine is the machine shown at the left, which was found by the German mathematicians Heiner Marxen and Jürgen Buntrock in 1990. This machine generates an astonishing total of 4098 1s before halting. In March 2001, the same team discovered a six-state machine that generates more than \( 10^{965} \) 1s—a number so large that a complete simulation on the fastest computers would take far longer than the expected lifetime of the universe. There may be six-state machines that produce even larger numbers of 1s, but it is clear that \( BB(k) \) grows extremely quickly as \( k \) increases.
Figure 9-1: Complete list of all 64 one-state Turing machines and their output on blank tape.
Useful properties of the $BB$ function

Before setting out to explore the philosophical implications of the function $BB(k)$, it is important to make two observations about its general nature:

1. *The function $BB(k)$ exists and is well-defined.* For any given number of states, there is a finite number of possible Turing machines. In a two-state machine, for example, each of the four instructions has two possible values (0 and 1) for the new symbol, two possible directions of motion (L and R), and three possible values for the new state (0, 1, and 2). Thus, the total number of 2-state Turing machines is
   \[(2 \times 2 \times 3)^4\]
   or 20,736. Using similar logic, the number of Turing machines with $k$ states can be expressed as follows:
   \[(2 \times 2 \times (k + 1))^{2k}\]
   This number, of course, can be extremely large, but there are still only a finite number of machines to consider of a certain size. Some of those machines will terminate when run on blank tape; others will get caught in an infinite loop and run forever. If you consider only the terminating machines, there must be some maximum value for the number of 1s, although it may be achieved by more than one machine.

2. *The function $BB(k)$ is a strictly increasing function of $k.* Adding a new state to a Turing machine program always enables you to generate more 1s than was possible with the smaller machine. Thus, for all values of $k$
   \[BB(k) < BB(k + 1)\]
   To see that this relationship must hold, think about how you might go about using an additional state. If nothing else, you could always construct the machine with $k + 1$ states by taking the best $k$-state Busy Beaver machine and composing it with the $M_{++}$ machine shown on the left. By definition, the original $k$-state machine writes $BB(k)$ 1s. Because the new machine does the same thing and then goes on to write an additional 1 in the first available space, the new machine writes a total of $BB(k) + 1$. Thus, it is always possible to get at least one more 1 if you have an additional state.

   It is important to note that the $M_{++}$ machine does not represent a mathematical function because it does not leave the tape head sitting on the leftmost 1. Since it does not produce a numeric value, it would be incorrect to compose another
machine after \( M^{++} \). On the other hand, if \( M^{++} \) is used as the final machine in a sequence, it will always write one more 1 than the machine without that extra state.

**Proving that the BB function is uncomputable**

The Busy Beaver function, however, has another property beyond those listed in the preceding section. Despite the fact that \( BB(k) \) must exist for any value of \( k \), it turns out that there is no Turing machine—and if you believe the Church-Turing thesis, no algorithmic process—that would allow you to compute in all cases what \( BB(k) \) is. What’s perhaps even more surprising is that it is possible to prove that no such computational method exists. Functions for which no computational strategy can exist are said to be uncomputable.

The task of this section is to prove the following theorem:

**Busy Beaver theorem**: \( BB(k) \) cannot be computed by any Turing machine.

The easiest way to establish the truth of this proposition is to employ a general strategy called **proof by contradiction**. In a proof by contradiction, you begin by assuming that the theorem is false and then show that such an assumption leads to a logical contradiction. Thus, in order to establish that \( BB(k) \) cannot be computed by a Turing machine, the first step is to make the following contrary assumption:

**Assumption**: \( BB(k) \) can be computed by some Turing machine.

The goal is to show that this assumption leads to an impossible conclusion.

The first step toward deriving the contradiction is to understand the implications of the assumption. If \( BB(k) \) is a computable function, there must be a specific Turing machine \( M_{BB} \) that starts with \( k \) 1s on the tape and writes out a sequence of 1s whose length gives the value of the function \( BB(k) \). Thus, if you invoked \( M_{BB} \) on the input tape

\[
\cdots \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ \cdots
\]

the program should run for a while—presumably quite some time, given that you know from your own experiments that finding the best Busy Beaver machine is a difficult problem—and then produce the following result

\[
\cdots \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ \cdots
\]
indicating that $BB(3)$ is 6. If you started the $M_{BB}$ machine again on an input tape with six 1s, it would churn—presumably for an even longer time—and write out a huge string of 1s equal to $BB(6)$.

The most important result that follows from postulating the existence of a machine $M_{BB}$ is that, if such a machine exists, it must have a finite number of states. Let’s call that number $\beta$. Just from playing around with the problem, you certainly must have the sense that any machine that could solve the general Busy Beaver problem must be extremely complex algorithmically, which means that $\beta$ is likely to be very large. Nonetheless, $\beta$ must be finite, since it is the number of states in a Turing machine that the proof-by-contradiction strategy allows us to assume exists.

From here, the rest of the proof consists of constructing a new Turing machine that leaves us with a contradiction. That machine is pieced together by composing Turing machines, including $M_{BB}$ along with several other machines that were already introduced in Chapter 8, such as $M_{2x}$ and $M_{+3}$.

It is useful to note that the $M_{+3}$ machine is representative of a more general family of Turing machines that add some constant to the input value on the tape. The seven-state machine $M_{+7}$ shown at the right, for example, adds 7 to its input. You can use $M_{+3}$ or $M_{+7}$ as a template to build a $k$-state machine $M_{+k}$ that adds $k$ to its input for any $k$ greater than 1. In particular, you can easily construct a machine $M_{+\beta}$ with $\beta$ states that adds $\beta$ to its input.

To establish the contradiction needed to complete the proof of the Busy Beaver theorem, all you need to do is construct a machine $M_?$ by composing the following five machines:

$$M_{+\beta} \rightarrow M_{+7} \rightarrow M_{2x} \rightarrow M_{BB} \rightarrow M_+$$

You then need to ask yourself the following questions:

1. How many states does $M_?$ have?
2. How many 1s does $M_?$ generate when started on blank tape?

To answer the first question, all you have to do is count up the number of states in the individual machines. $M_{+\beta}$ has $\beta$ states, $M_{+7}$ has 7, $M_{2x}$ has 6, $M_{BB}$ (if it were to exist) has $\beta$ states by definition, and $M_+$ has 1. The total number of states in the machine is therefore $2\beta + 14$.

To answer the second question, you need to follow the logic of the machine, applying each of the functions implemented by the Turing machine to the previous result. The first two machines write out a total of $\beta + 7$ 1s on the tape. The machine $M_{2x}$ then doubles this number so that the tape contains a total of $2\beta + 14$ 1s. By
hypothesis, $M_{BB}$ computes $BB(2\beta + 14)$, which is the maximum finite number of 1s that can be produced using a machine with $2\beta + 14$ states. The final $M_{++}$ machine adds one more 1 to that result. Thus, $M_? writes$ $BB(2\beta + 14) + 1$ 1s on blank tape. These calculations are summarized in the following table:

<table>
<thead>
<tr>
<th>Machine</th>
<th>States</th>
<th>Number of 1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{\beta}$</td>
<td>$\beta$</td>
<td>$\beta$</td>
</tr>
<tr>
<td>$M_7$</td>
<td>7</td>
<td>$\beta + 7$</td>
</tr>
<tr>
<td>$M_{26}$</td>
<td>6</td>
<td>$2\beta + 14$</td>
</tr>
<tr>
<td>$M_{BB}$</td>
<td>$\beta$</td>
<td>$BB(2\beta + 14)$</td>
</tr>
<tr>
<td>$M_+$</td>
<td>1</td>
<td>$BB(2\beta + 14) + 1$</td>
</tr>
<tr>
<td>$M_?</td>
<td>2\beta + 14</td>
<td>BB(2\beta + 14) + 1</td>
</tr>
</tbody>
</table>

There is, however, a serious problem here. $M_?$ has only $2\beta + 14$ states, and yet it writes $BB(2\beta + 14) + 1$ 1s on blank tape. Somehow, the process of constructing $M_?$ has produced a machine that writes more 1s on blank tape than should be possible for a machine of that size. This result is a logical contradiction. The only questionable step in the process is the initial assumption that $M_{BB}$ exists. If it does, the remainder of the proof follows logically, including the impossible contradiction. As a result, you know that $M_{BB}$ cannot exist, which completes the proof of the Busy Beaver theorem.

**Busy Beaver and the Church-Turing thesis**

The fact that $BB(k)$ cannot be computed by a Turing machine takes on additional significance when you consider this result in light of the Church-Turing thesis described in section 8-4. Because the Turing machine provides what computer scientists believe to be a universal model of computation, the fact that no Turing machine can compute the Busy Beaver functions means that no other computational paradigm—not Babbage’s Analytical Engine, not a modern computer running a JavaScript program, indeed not a program in any language at all—could perform that computation either.

If this claim seems hard to believe, the following argument may help. At some level, the Church-Turing thesis remains a hypothesis. There is no way to be certain that no one will ever come up with a computation model that is more powerful than a Turing machine. What is true, however, is that none of the models that people have yet devised are any more powerful than the extraordinarily primitive machine that Turing devised. In particular, it is possible—admittedly tedious, but certainly possible—to simulate any modern programming language like JavaScript using a Turing machine. Given that fact, the proof that $BB(k)$ is uncomputable certainly must apply to JavaScript. If there were a JavaScript program to compute $BB(k)$,
then all you would have to do is simulate the operation of that program on a Turing machine, which would then give you a Turing machine program to compute that same function. But the proof of Busy Beaver theorem makes it clear that no such Turing machine can exist, which implies that no JavaScript program can compute \( BB(k) \).

**Confronting the nagging doubts**

*This proof, although perfectly sound, has the disadvantage that it is likely to leave the reader with a feeling that “there must be something wrong.”*


If you are like most students of computer science, the proof of the Busy Beaver theorem will not convince you right away. Most people are left with nagging doubts, partly because it really doesn’t seem so hard—at least in theory—to compute \( BB(k) \). There are, after all, only a finite number of machines of any given size. If you wanted to compute \( BB(k) \) for any particular value of \( k \), what would be wrong with the following strategy:

1. Generate every possible \( k \)-state Turing machine in some systematic order.
2. For each machine, simulate its operation and count how many 1s it produces.
3. Report the largest number of 1s generated as the value of \( BB(k) \).

This strategy is, after all, precisely the technique used in Figure 9-1 to find \( BB(1) \). The Busy Beaver theorem tells us that no algorithmic strategy can solve this problem, so there must be something wrong with this approach. Understanding just what the problem is, however, turns out to be rather subtle.

The fundamental problem with this strategy is that there is no effective way to carry out step 2. Simulating a Turing machine is easy enough, as long as the machine you’re simulating halts. What happens if you simulate a machine that goes into an infinite loop? Unless you could figure out that the machine was looping and stop the simulation process, step 2 would take forever, just as the machine it is simulating does. This observation sets the stage for one of the fundamental discoveries of computer science, which is described in the section that follows.

### 9.2 The halting problem

Although the Busy Beaver problem is fun in the way that challenging puzzles often are, it hardly seems important in any fundamental sense. If the only examples of uncomputable functions are as esoteric as \( BB(k) \), the concept of uncomputable functions would not have a great deal of relevance to the practice of computer
science. As it happens, many practical problems turn out to be uncomputable as well. Of these, the most well known example is the halting problem, which can be expressed in the context of Turing machines as follows:

Given a Turing machine and its input, it is impossible to determine whether the machine will halt.

Like the Busy Beaver problem, the halting problem is unsolvable.

There is, however, nothing special about Turing machines. Because Turing machines have the same computational power as any existing computational process, the problems that are unsolvable in the Turing machine world will also be unsolvable in any programming language. Thus, the halting problem can be expressed equally well in the following form

Given a program in any programming language and its input data, it is impossible to determine whether that program will halt.

At this point, the implications get more interesting. It is impossible, for example, to write an automatic debugging system that will look at a program and give you a reliable answer as to whether that program contains an infinite loop.

**Busy Beaver and the halting problem**

In mathematics, one of the easiest ways to prove a new theorem is to transform it into an existing theorem that has already been proved. That process is called reduction. For example, you can quickly prove that the halting problem is unsolvable by using the fact that the Busy Beaver problem is already known to be unsolvable. If assuming that a solution to the halting problem exists contradicts the Busy Beaver theorem, that assumption cannot be justified.

If you could solve the halting problem, it would be possible to compute $BB(k)$ using the following straightforward adaptation of the strategy presented earlier in this chapter:

1. Generate every possible $k$-state Turing machine.
2. For each machine, use the solution to the halting problem to determine whether that machine halts on blank input tape. Ignore any machine that goes into an infinite loop.
3. Simulate the operation of any remaining machine and count how many 1s it produces.
4. Report the largest number of 1s generated as the value of $BB(k)$. 
The addition of the test for whether the machine halts fixes the problem that rendered this solution ineffective before. This construction allows you to make the following argument:

1. If it were possible to solve the halting problem, it would then be possible to compute $BB(k)$.
2. The proof of the Busy Beaver theorem shows that it is impossible to compute $BB(k)$.
3. Therefore, it must not be possible to solve the halting problem.

The halting problem in JavaScript

The proof of the halting problem presented in the preceding section will not be at all convincing if you are still unsure about the Busy Beaver theorem. It is, fortunately, possible to prove that the halting problem is unsolvable without any recourse to Turing machines. We’ll look instead at programs in JavaScript and ask whether there might not be a way to test whether those programs halt.

Once again, we’ll attempt a proof by contradiction and assume that it is possible to write a program that makes such a determination. If so, it must be possible to write a predicate function called $doesProgramHalt$ that takes the name of a JavaScript program file and returns $true$ if that program halts and $false$ if it goes into an infinite loop. For example, if you had such a function you could use it to report the status of a program using the following code:

```javascript
if (doesProgramHalt(filename)) {
    Console.println("The program halts.");
} else {
    Console.println("The program runs forever.");
}
```

This code is perfectly fine if the function $doesProgramHalt$ exists. The problem, however, lies in the fact that one could easily change these lines to produce the Paradox.js program shown in Figure 9-2. The new version of the code inside the if statement looks like this:

```javascript
if (doesProgramHalt("Paradox.js")) {
    Console.println("The program runs forever.");
    while (true) {
        /* Loop forever doing nothing */
    }
} else {
    Console.println("The program halts.");
}
```
What happens if you run the paradox function in the file Paradox.js? The first step in the main program calls the function doesProgramHalt. As you can see from the comments, doesProgramHalt analyzes the code in the file it is given and simulates its operation, seeking to determine whether it halts. In this case, filename is "Paradox.js", which means that doesProgramHalt has to scan the code for Paradox.js and figure out what the first function in that program does.

If you think about this situation a bit, you will discover that doesProgramHalt cannot possibly return the correct answer. If its analysis determines that the program contained in the file Paradox.js halts, then doesProgramHalt will return true. In that case, however, the code in the main program executes the loop
while (true) {
    /* Loop forever doing nothing */
}

which causes it enter an infinite loop. That loop contradicts the result of 
doesProgramHalt, which claimed that Paradox.js halts. On the other hand, 
doesProgramHalt cannot return false to indicate that the Paradox.js loops. If 
it does, the code executes a single Console.println call that once again 
contradicts the result that doesProgramHalt returns. If the function reports that 
Paradox.js halts, it loops; if it reports that Paradox.js loops, it halts.

Clearly, something is wrong. The only questionable assumption is that is 
possible to write a function like doesProgramHalt that can analyze any program 
and determine whether it terminates. If such a function were possible, then it would 
surely fail for certain programs such as Paradox.js. Because this failure 
establishes a contradiction, it must be impossible to write a general implementation 
of doesProgramHalt that works in all cases.
The purpose of this chapter is to introduce you to the most significant open question in theoretical computer science. Although some of the terminology is undoubtedly cryptic at the moment, the $P = \text{NP}$ question can be stated quite simply as follows:

Let $P$ be the class of all decision problems that can be solved by a standard deterministic Turing machine in polynomial time and $\text{NP}$ be the similarly defined class of decision problems that can be solved by a nondeterministic Turing machine in polynomial time. Are the classes $P$ and $\text{NP}$ the same?

In many ways, this question is the computer science counterpart to such famous problems in mathematics as the Four-Color Problem (proposed in 1852 by Francis Guthrie, proved in 1976 by Kenneth Appel and Wolfgang Haken) or Fermat’s Last Theorem (proposed in 1637 by Pierre de Fermat, proved in 1995 by Andrew Wiles). There is, however, a critical difference in the potential significance of the result. While the proofs of the Four-Color Problem and Fermat’s Last Theorem generated considerable excitement in the mathematics community—and even a fair amount of interest among the broader segment of the public interested in mathematical results—they had little or no practical effect on other disciplines. Settling the question as to whether $P = \text{NP}$—particularly if these two classes turn out to be the same—would have profound scientific and practical implications.

The importance of this problem is underscored by the fact that a private mathematical foundation has established a prize of $1,000,000 for its solution, as described in the box on the following page.
P = NP? — The million dollar question

Of the theoretical ideas considered in this book, the question of whether $P = NP$—or whether the class of problems solvable in polynomial time is identical to the class of problems solvable in polynomial time using a nondeterministic process—is one of the most fascinating. For one thing, the problem is still unsolved. For another, a solution to this problem would have major practical importance. And for a third, anyone who solves this problem stands to win a million-dollar prize from the Clay Mathematics Institute, a private foundation based in Cambridge, Massachusetts. The $P = NP$ question is the only problem from theoretical computer science to make their list.

The $P = NP$ problem was first raised in 1971 in a now classic paper by Steven A. Cook, professor of computer science at the University of Toronto. In that paper—the main results of which are outlined in this chapter—Cook established a fundamental property linking a large class of computational problems: that an efficient solution to any of those problems implies the existence of efficient solutions for the others. Because many of these problems have significant commercial importance, interest in Cook’s result was high.

As the discussion in this chapter makes clear, the implications of a solution to the $P = NP$ problem, no matter which way it turns out, would be profound. If the two classes are shown to be the same, many practical problems that now appear quite difficult would have relatively simple solutions. Conversely, proving that these two classes are different—as indeed most computer scientists believe—would increase our confidence in the security of various cryptographic systems that depend on the difficulty of solving problems in the class NP. These issues are discussed in more detail in Chapter 12.

Beyond the practical consequences of the $P = NP$ debate, the problem also provides opportunity for playful speculation. The concept of NP-completeness turns up in surprising contexts beyond the cryptographic examples in this chapter, including origami, protein folding, and the Minesweeper computer game.
10.1 Refining the definitions of P and NP

Before you can understand the \( P = NP \) question, you need to have a deeper appreciation of what the question means. The definitions of the classes \( P \) and \( NP \) given in the preceding section, while reasonably concise, depend on the definition of such terms as decision problem, polynomial time, and nondeterministic. The sections that follow define each of these terms.

**Decision problems**

In the definition of \( P \) and \( NP \), the easiest part to unravel is the concept of a decision problem, which refers to a problem for which the solution is always either true or false. For example, determining whether an integer is prime is an example of a decision problem; finding the actual factors of a composite number is not.

In the context of Turing machines, there are several ways to represent decision problems. One common approach is to use the current symbol to represent the result of the computation, where a 1 on the final tape square represents true and a 0 represents false. For the examples in this chapter, however, it is easier to define two new instructions—accept and reject—that can be included as part of the state table. If a Turing machine executes an accept instruction, the result is true; conversely, if a machine executes a reject instruction, the result is false. In either case, computation ends immediately after executing the instruction, without making any changes to the contents of the tape or the position of the tape head.

As an example, you can use these new instructions to create a Turing machine \( M_{odd} \) that decides if its input is odd, as follows:

This machine moves rightward over the input, going back and forth between states 1 and 2. If it reaches the end of the number in state 1, the number must have been even, so the machine rejects the input. If the end of the number shows up in state 2, there must have been an odd number of 1s, so the machine accepts.

**Polynomial time**

In class—and in a forthcoming chapter—you will have the chance to learn a little about the idea of algorithmic efficiency. A polynomial-time algorithm is one whose complexity is bounded by a polynomial function of the size of the problem. For example, algorithms whose computational complexity is \( O(N) \) or \( O(N^2) \) or even \( O(N^{10}) \) are all said to run in polynomial time. In general, problems that can be
solved in polynomial time are considered to be *tractable*, even though the running time may be very large. By contrast, problems for which no polynomial-time algorithm exists are regarded as *intractable*.

The most common examples of nonpolynomial algorithms are those that run in exponential time, which is expressed in big-O notation as $O(2^N)$. For small values of $N$, $2^N$ will be smaller than a high-order polynomial. For example, if $N$ is 10, $2^N$ is 1024, while $N^{10}$ is 10,000,000,000. As $N$ grows larger, however, $2^N$ eventually becomes larger than any polynomial expression. For the functions $2^N$ and $N^{10}$, the crossover occurs when $N$ hits 59. For all values of $N$ that are 59 and above, $2^N$ will be larger than $N^{10}$. The difference, moreover, grows rapidly. When $N$ is 100, for example, $N^{10}$ is $100,000,000,000,000,000,000,000$; by this point, however, $2^N$ is $1,267,650,600,228,229,496,703,205,376$, which is more than a billion times as large.

**Nondeterminism**

Many problems—and particularly those for which the simplest algorithms run in exponential time—are characterized by a branching set of choices, similar to that you would encounter in solving a maze. When you reach the first choice point, you must make a decision about which way to go. If your choice turns out to be a blind alley, you will have to retrace your steps and make a different choice. As long as you have to backtrack, the process is slow. If you could, however, explore every possible path simultaneously, you could solve such problems in much less time.

To make this idea more concrete, suppose that you were designing a system to solve a maze, such as the following, in which the beeper indicates the goal:

![Maze Diagram](image)

If Karel wants to solve this maze, it must be able to apply some systematic solution strategy. One strategy that works for any maze is to have Karel explore every
possible path recursively, backtracking to the most recent decision point whenever it encounters a dead end. The simplest recursive solutions of this form have exponential running times in the worst case, because the number of paths can grow exponentially with respect to the size of the maze.

But consider for a moment what would happen if Karel were somehow able to create a clone of itself at any time it wished. If that were true, it could simply clone itself at every decision point and have an ever-expanding set of robots explore all possible paths in parallel. Thus, after walking to the first decision point, Karel would construct a clone. The clone could explore one path while the original Karel explored the other, like this:

At each new decision point, the same cloning operation takes place. If any robot reaches the exit, the problem is solved. In this maze, for example, you end up with the following collection of Karel clones, most of which are stuck in dead ends, but one of which has reached its goal:
Note that the cloning strategy always solves a maze in a number of steps equal to the length of the shortest path, which is clearly no larger than the size of the maze. Thus, the cloning strategy is linear—and therefore polynomial—in its complexity.

The cloning algorithm for solving a maze is an example of a parallel strategy in which the computation for each possible solution path can be carried out simultaneously. Another way of thinking about such algorithms is to imagine that, instead of cloning itself, Karel is somehow able to guess the correct path at every decision point. While it is certainly hard to understand where one would ever find a robot that always guesses correctly, such a strategy would be exactly as effective as the cloning one. If your goal is to determine how long it takes any of the Karel clones to solve a maze, you will end up following the one that takes the correct path at every opportunity. Because the parallel try-every-path strategies are equivalent in computational complexity to those that can simply guess the correct path, you might as well assume that your algorithms can always make the correct guess whenever they face a choice. Strategies that involve making guesses at decision points are said to be nondeterministic.

**Nondeterministic Turing machines**

The last issue that you need to understand before turning back to the \( \text{P} = \text{NP} \) question is how nondeterminism applies to Turing machines. The traditional Turing machine represents a model for deterministic algorithms, but it is easy to modify the basic model so that it can represent nondeterministic algorithms, in which the machine can pursue several different computations simultaneously. A **nondeterministic Turing machine** is the same as a standard Turing machine except that there may be more than one instruction for each combination of a current state and current symbol. For example, a nondeterministic Turing machine might include an instruction of the following form:

If this machine is in state 1 scanning a 0, the program can *either* write a 1, move right, and enter state 2 *or* write a 0, move left, and stay in state 1. If either choice has the machine end up in an accepting state, then the nondeterministic machine accepts its input. The nondeterministic machine rejects its input only if *all* possible computations end up in a rejecting state or fail to terminate.

As in the discussion of the cloned-robot strategy described in the preceding section, you can think about nondeterministic Turing machines either as operating in parallel, pursuing every possible option at once, or simply as machines that are able to guess correctly at every opportunity. These two models turn out to be equivalent, and you should use whichever formulation feels most natural.
As a concrete example of a nondeterministic Turing machine, consider the Turing machine $M_{2,3}$, which determines whether its input is divisible by either 2 or 3:

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>accept</td>
</tr>
<tr>
<td>1</td>
<td>1R2, 1R4</td>
</tr>
<tr>
<td>2</td>
<td>reject</td>
</tr>
<tr>
<td>3</td>
<td>1R2</td>
</tr>
<tr>
<td>4</td>
<td>reject</td>
</tr>
<tr>
<td>5</td>
<td>1R4</td>
</tr>
<tr>
<td>6</td>
<td>accept</td>
</tr>
</tbody>
</table>

This program works by splitting the computation into two separate parts, one that checks whether the number is divisible by 2 (states 2 and 3) and one that checks whether the input is divisible by 3 (states 4, 5, and 6). State 1 sends the machine along each of these paths. The instruction $1R2$ initiates the computation that checks for divisibility by 2, and the instruction $1R4$ starts checking for divisibility by 3. Along each of these two paths, the machine moves to the right across a sequence of 1s, either in groups of two or groups of three.

As an example, suppose you start $M_{2,3}$ on a tape containing the input value 4, represented by the following tape:

\[ \cdots 0 1 1 1 1 0 \cdots \]

Since $M_{2,3}$ is in state 1 scanning a 1 on the tape, the machine must execute a nondeterministic operation, which is essentially both $1R2$ and $1R4$. One way to think about this operation is that the machine clones itself before executing these instructions. After creating a copy of itself, one of the machines executes the instruction $1R2$ while the other goes off and executes $1R4$. Thus, after executing this instruction, there are two machines running in parallel, as shown in Figure 10-1. After the first cycle, the machines operate independently. The machine diagrammed on the left alternates between states 2 and 3, skipping 1s in pairs as it goes. The machine on the right cycles through states 4, 5, and 6 as it skips over 1s in groups of three. When the computation reaches the end of the input, the machine on the left is in state 3 and therefore accepts its input. The machine on the right, meanwhile, has reached the end in state 4, which means that it rejects. These results are certainly what one would expect, given that 4 is divisible by 2 but not by 3.

According to the definition of acceptance for a nondeterministic Turing machine, $M_{2,3}$ accepts the input because at least one of its clones does. Were you to run this machine on a tape containing five 1s, both of the clone machines would end up in
rejecting states, which means that the composite nondeterministic machine would reject the input.

**P = NP reconsidered**

At this point, you have everything you need to understand the statement of the $P = NP$ question given at the beginning of this handout. The class $P$ is the set of all decision problems that can be solved in polynomial time on a standard deterministic Turing machine. The class $NP$ is similarly defined as the set of all decision problems that can be solved in polynomial time on a *nondeterministic* Turing machine. The $P = NP$ question asks whether these two classes are the same. Is it the case that every problem that can be solved in polynomial time by a nondeterministic machine can also be solved in polynomial time by a deterministic one? Or are there some problems that can be solved in polynomial time only if nondeterministic techniques are used?

Intuitively, it seems as if the power of nondeterminism—of being able to guess the correct path at every opportunity—would be enormously powerful, substantially reducing the complexity of a problem. At first glance, the maze-solving example
seems to be a case in point. The straightforward recursive strategy for solving a maze requires exponential time, while the corresponding nondeterministic strategy is linear. On the other hand, it is easy to design a deterministic algorithm that solves a maze in linear time. All you have to do is mark the squares you’ve visited and never retrace your path. Thus, in the case of solving a maze, the extraordinary power of being able to guess correctly does not change the computational complexity in any fundamental way. You can find algorithms that do just as well without ever having to rely on nondeterminism.

The same situation applies to the case of the Turing machine that decides whether a number is divisible by 2 or 3. While this problem can certainly be solved in linear time by the nondeterministic Turing machine defined in the preceding section, it can be solved just as quickly by the following deterministic machine:

```
0 1
1 accept 1R2
2 reject 1R3
3 accept 1R4
4 accept 1R5
5 accept 1R6
6 reject 1R1
```

At this point, it makes sense to ask whether nondeterminism ever leads to a fundamental increase in computational efficiency. Are there problems in which the power of nondeterminism makes a critical difference, taking a problem out of the intractable exponential domain and moving it back into the scope of tractable, polynomial-time problems? This question lies at the heart of the $P = NP$ debate.

### 10.2 The traveling salesman problem

Before moving on to look at the theoretical issues that underlie the $P = NP$ debate, it makes sense to introduce a couple of problems from the class $NP$, so that you have some intuition about how you might go about solving them. Suppose that you are part of the sales force for a business and need to find a way to start at your home office, visit a number of cities, and return home, in a way that minimizes the cost of travel. For example, suppose that you need to visit each of the six cities shown in Figure 10-2, starting in Atlanta and then returning to Atlanta at the end.

In the route map shown in Figure 10-2, there are only a small number of circuits one can make. There are a total of eight different paths, of which four are simply the reversed forms of the others. The distinct paths and their associated costs are as listed at the bottom of the figure in increasing order of cost. The best path is therefore the first one, which avoids the cost of the expensive Boston-Eugene flight.
If your goal is to identify the order of the cities along the cheapest possible route, this problem is not a decision problem because it does not have a simple yes or no answer. Another way of looking at the problem, however, is to imagine that you have some fixed amount of money to spend and all you need to know is whether you can make the trip within that budget. For example, if your budget were $2500, there are two possible routings that qualify, so the answer to the question would be yes. If you only had $1500, you’d be stuck. When it is phrased in this way, the problem of finding a circular path whose cost is less than a certain threshold becomes a legitimate decision problem. In computer science, that problem is called the **traveling salesman problem**, or **TSP** for short.

As far as anyone knows, there is no deterministic algorithm that solves the traveling salesman problem in polynomial time. It is, however, easy to generate a nondeterministic solution. If you can follow multiple paths simultaneously—or,
equivalently, guess correctly at every decision point—you can solve the problem in linear time. Thus, the traveling salesman problem is in the class \( \textbf{NP} \). In the deterministic world demanded for inclusion in the class \( \textbf{P} \), however, the best-known algorithms for the traveling salesman problem have exponential performance and are equivalent in computational efficiency to generating all possible routes and checking the cost. The number of possible routes in a connected network of \( N \) cities grows in proportion to \( 2^N \), which gives rise to the exponential behavior. At the same time, no one has yet been able to prove that no polynomial-time algorithm for this problem exists. There might be some clever algorithm that makes this problem tractable. Most computer scientists believe that no such algorithm exists, but no one has yet been able to establish that result.

If someone does come up with a polynomial solution to the traveling salesman problem, the effect on practical computation would be enormous. The obvious implication, of course, is that it would then be computationally feasible to generate optimal routes for a salesman traveling through a series of cities. Finding a practical solution to the traveling salesman problem, however, would have impact far beyond the specific application of planning optimal routes. In fact, if anyone finds a polynomial time solution to the traveling salesman problem, then every problem in the class \( \textbf{NP} \) will also have a polynomial-time solution, which in turn means that the classes \( \textbf{P} \) and \( \textbf{NP} \) are in fact the same.

The traveling salesman problem belongs to a special subset of \( \textbf{NP} \) called **\( \textbf{NP} \)-complete problems**, which are at least as hard to solve as any other problem in the class. More formally, a decision problem is \( \textbf{NP} \)-complete if (1) it is in the class \( \textbf{NP} \) and (2) it is possible to prove that if that problem has a polynomial time solution, than every problem in \( \textbf{NP} \) must also have a polynomial-time solution.

The concept of \( \textbf{NP} \)-completeness first appeared in a 1971 paper by Stephen Cook, professor of computer science at the University of Toronto. In this paper, which had the title “The Complexity of Theorem-Proving Procedures,” Cook proved that if a polynomial-time solution exists for the satisfiability problem discussed in the following section, then a polynomial-time solution exists for every problem in the class \( \textbf{NP} \). Since the appearance of Cook’s initial proof in 1971, thousands of other problems have been shown to be \( \textbf{NP} \)-complete. Some of the more interesting problems are outlined in Figure 10-3.

### 10.3 Satisfiability

The first problem shown to be \( \textbf{NP} \)-complete is the **satisfiability problem** (often shortened to the acronym \( \text{SAT} \)), which asks whether it is possible to assign values to the variables of a logical expression so that the entire expression is true. The logical expression must be in a specific form, as described in the sections that follow.
Subset sum. Suppose that you have a set of integers called $S$. The subset-sum problem asks whether there is a subset of the elements of $S$ that add up to a particular target value $t$. For example, if $S$ is the set $\{-3, 5, 7, 10\}$, the subset-sum problem when $t$ is 12 returns the answer true because the elements in the subset $\{-3, 5, 10\}$ add up to 12. By contrast, if $t$ were 11, the answer is false because it is impossible to choose a subset of $S$ whose values adds up to 11. In his early study of NP-complete problems in 1972, Richard Karp proved that the subset-sum problem is NP-complete, although his original papers refers to the problem by a different name.

Graph coloring. Suppose that you have a graph consisting of a set of vertices connected by a set of edges, which might, for example, look like either of the graphs to the right. The vertices of the top graph can be colored using three colors so that no two vertices connected by an edge share the same color. You could, for example, color nodes 1 and 3 white, 2 and 4 gray, and 5 blue. In the bottom graph, however, all four nodes are interconnected, which means that each must each have a different color. Deciding whether a graph can be colored with $k$ colors is NP-complete, which was also established by Richard Karp in his 1972 paper. Graph coloring has several practical applications in computer science. In particular, graph coloring provides the basis for the standard algorithms used to assign variables to hardware registers when compiling a program.

Origami folding. The diagram at the right shows the first eight folds on the way to the creation of a classic origami crane. In some of these folds, the crease rises toward you from the paper. These are called mountain folds and appear in the diagram as dashed lines. In other folds, the crease moves away from you. These are called valley folds and appear as dotted lines. In 1996, Marshall Bern and Barry Hayes proved that deciding whether a particular pattern of mountain and valley folds will produce a flat origami figure is NP-complete. More recently, computational biologists have used similar techniques to prove that the problem of folding certain protein structures into a fixed volume is also NP-complete.

Minesweeper. One of the most widely publicized problems in the NP-complete domain is that of determining whether a particular pattern of warning counts in the popular Microsoft Minesweeper game is consistent. In 2000, Richard Kaye published a paper proving that determining the minesweeper consistency problem is NP-complete. Because of the popularity of the game, Kaye’s result was reported in newspapers and magazines throughout the world. In 2011, Allan Scott, Ulrike Stege, Iris van Rooij reported an error in Kaye’s original proof of NP-completeness, but were able to verify his conclusion that the minesweeper consistency problem is algorithmically hard.
Propositional logic

The framework in which the satisfiability problem resides is called propositional logic, which is a branch of mathematics concerned with the truth or falsity of abstract statements. The discipline of propositional logic was formalized by George Boole, who also gave his name to Boolean data, as you know from Chapter 2. Boole’s propositional logic is similar to work by the Greek philosopher Chrysippus in the 3rd century BCE and by Gottfried Wilhelm von Leibniz in the 17th century.

In propositional logic, the fundamental unit out of which expressions are built is called a proposition, which is any statement whose value is either true or false. Individual propositions are conventionally assigned single-character names such as $p$ and $q$. You could, for example, assign the name $p$ to the proposition “it is raining,” and the name $q$ to the proposition “there are clouds in the sky.”

Propositional logic allows you to combine individual propositions into larger expressions using operators that express logical relationships of precisely the sort used by the logic gates in Chapter 5. For example, propositional logic uses the operator $\land$ to express the English word and, the operator $\lor$ to express the English word or, the operator $\neg$ to express the English word not, and the operator $\rightarrow$ to express the idea behind the English phrase if . . . then. Thus, given the propositions defined in the previous paragraph, the expression $p \rightarrow q$ has the same meaning as the following English sentence:

If it is raining, then there are clouds in the sky.

The primary advantage of using propositional logic is that doing so allows you to exploit the mathematical properties of logical expressions independent of the interpretations that the individual propositions might have at any given moment. Given any propositions $p$ and $q$, the truth of expressions like $p \land q$, $p \lor q$, $p \rightarrow q$, and $\neg p \lor q$ can be determined directly by applying the rules of logic to the values of the propositions $p$ and $q$. These rules can be summarized in the form of a truth table, which is simply a matrix that shows the value of one or more output expressions as a function of the values of the individual terms. The truth table for these three expressions looks like this:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \land q$</th>
<th>$p \lor q$</th>
<th>$p \rightarrow q$</th>
<th>$\neg p \lor q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

The fact that the last two columns are identical shows that the expressions $p \rightarrow q$ and $\neg p \lor q$ are logically identical, which makes it possible to replace appearances of the $\rightarrow$ operator with its expanded form.
The satisfiability problem consists of determining whether any assignment of
truth values to an expression in propositional logic. As a simple example, suppose
that you want to determine whether the following expression is satisfiable:

\[ p \land \neg q \]

In this case, it is easy to find assignments to the variables \( p \) and \( q \) that make the
whole expression true. All you have to do is set \( p \) to true and \( q \) to false. If, however, the expression were instead

\[ p \land \neg p \]

you’d be stuck, because there is no way that the variable \( p \) can be true and false at
the same time. This second expression is therefore unsatisfiable.

**Conjunctive normal form**

The formal definition of the satisfiability problem requires that logical expressions
be written in a conventional style called **conjunctive normal form** or **CNF**, which is
defined by the following rules:

- Each expression in conjunctive normal form consists of a list of **conjuncts**
  connected by \( \land \), which represents the logical and operator.
- Each **conjunct** is a list of **terms** connected by \( \lor \), which representing the logical or
  operator.
- Each **term** consists of a variable name, optionally with a negation sign, which is
  conventionally indicated by a horizontal bar across the top of the variable name.

The following expression, for example, is in conjunctive normal form:

\[ (p \lor \overline{q}) \land (q \lor r) \land (p \lor \overline{r}) \land \overline{p} \]

Given an expression in conjunctive normal form, you can determine whether that
expression is satisfiable by writing out a truth table for each possible assignment of
the variables. For this expression, the truth table (including the value of each of the
conjuncts) looks like this:

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( r )</th>
<th>( p \lor \overline{q} )</th>
<th>( q \lor r )</th>
<th>( p \lor \overline{r} )</th>
<th>( \overline{p} )</th>
<th>( (p \lor \overline{q}) \land (q \lor r) \land (p \lor \overline{r}) \land \overline{p} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
</tr>
</tbody>
</table>
As you can see, there is no assignment of truth values to the variables \( p, q, \) and \( r \) for which the last column in the truth table is true, which means that the expression is unsatisfiable.

### 10.4 Cook’s theorem

At first reading, the definition of \( \text{NP} \)-complete problems in the preceding section raises more questions than it answers. The defining characteristic of the problems in the \( \text{NP} \)-complete class is that finding a polynomial-time solution strategy for any one of them—out of the thousands of \( \text{NP} \)-complete problems that have been identified—implies that every problem in \( \text{NP} \) can be solved in polynomial time. How would one go about proving such a claim?

In his 1971 paper that introduced the notion of \( \text{NP} \)-completeness, Stephen Cook proved the following theorem:

**Cook’s theorem:** If a polynomial-time solution exists for the satisfiability problem, then a polynomial-time solution exists for every problem in the class \( \text{NP} \).

The overall strategy for the proof of Cook’s theorem looks like this:

1. **Assume that a polynomial-time solution exists for the satisfiability problem.** After all, Cook’s theorem is concerned only with what happens if a polynomial-time solution exists for satisfiability. If no such solution exists, there is nothing to prove. Thus, in trying to prove the rest of the theorem, you may assume you have a polynomial-time solution for satisfiability that you can then use as part of your argument.

2. **Select an arbitrary instance of a problem in the class \( \text{NP} \).** To prove Cook’s theorem, you need to show that the existence of a polynomial-time solution for satisfiability implies that any problem in \( \text{NP} \) can also be solved in polynomial time. You can start the process by assuming that you are working with some problem called \( X \) that you know comes from the class \( \text{NP} \). Within that problem domain, you may also assume that your immediate goal is to solve a particular instance of the problem, represented by a Turing machine tape containing the input \( Y \). Because \( X \) is in \( \text{NP} \), you know that there must be a nondeterministic Turing machine \( M_X \) that solves it in polynomial time. Moreover, the fact that \( M_X \) runs in polynomial time means that there must be some polynomial \( p_X \) that gives an upper bound on the number of steps that \( M_X \) needs to run. Since you have also chosen the input \( Y \), you know the size of the problem and can therefore determine how many steps will be required.

3. **Use a polynomial-time process to transform the problem instance into an equivalent satisfiability problem.** The essential step in the proof of Cook’s
theorem consists of demonstrating that any problem in \textbf{NP} can be recast as a satisfiability problem. To do so, you need to start with the nondeterministic Turing machine $M_X$ that solves the problem, along with the input value $Y$ representing a specific instance of that problem. The instructions in $M_X$ can be reformulated as an expression in propositional logic that simulates the operation of the Turing machine when executed on $Y$. The rules used to construct that expression force its value to be true if $M_X$ accepts $Y$, but false if $M_X$ rejects it. In other words, the logical expression is satisfiable if and only if the Turing machine from which it was derived accepts its input.

4. \textit{Invoke the polynomial-time solution for satisfiability on the transformed problem.} To solve the original problem, all you have to do is complete the transformation in step 3 and then use the satisfiability checker—which you can assume exists as described in step 1—to solve the resulting instance of a satisfiability problem. Because the transformation process runs in polynomial time, the complete solution will also run in polynomial time.

This strategy provides enough of an overview to give you a sense of how such proofs work but leaves out the details of step 3, which consists of transforming a nondeterministic Turing machine $M_X$ and its input $Y$ into a satisfiability problem. The basic idea behind that transformation is to define logical variables that represent the entire state of the machine at each point in all possible execution sequences of the machine. The next few sections outline the details of that process.

\textbf{Defining the logical variables}

The first step toward filling in the details of the proof is selecting a set of logical variables that are sufficient to record the current state of the Turing machine. For this purpose, you need three sets of variables, as follows:

1. A set of variables $s_{k,t}$ that correspond to the assertion that the Turing machine is in state $k$ at time $t$. In addition to the numbered states in the machine, the logical formula is easier to construct if you also have variables for the accepting and rejecting states, which are written as $s_{\text{acc},t}$ and $s_{\text{rej},t}$ for each time $t$.

2. A set of variables $p_{i,t}$ to indicate that the tape head is in position $i$ at time $t$. The tape squares are numbered in both directions from the initial position of the tape head, which is defined to be position 0. Thus, the variable $p_{-1,2}$ indicates that the machine is scanning the position immediately to the left of the original tape head position at time $t = 2$.

3. A set of variables $c_{i,t}$ to indicate the contents of tape square $i$ at time $t$. If the variable is true, the corresponding tape square contains a 1; if the variable is false, the tape square contains a 0. Positions on the tape are numbered exactly as they are for the $p_{i,t}$ variables.
These variables indicate the entire state of the Turing machine computation at each instant in time. They can, moreover, be used to create logical expressions that represent a particular state of the machine. For example, suppose that you were about to start an arbitrary Turing machine on the input tape

\[
\cdots 0 0 0 1 1 1 0 \cdots
\]

At time \( t = 0 \), just before the beginning of the first instruction, the Turing machine computation can be described in part by the logical expression

\[
s_{1,0} \land p_{0,0} \land \bar{c}_{-3,0} \land \bar{c}_{2,0} \land \bar{c}_{1,0} \land c_{0,0} \land c_{1,0} \land c_{2,0} \land \bar{c}_{3,0}
\]

If you read across this expression from left to right, it says that the machine starts in state 1, that the tape head is initially in position 0, that the three tape square to the left of the initial tape head contain 0s, that the next three positions contain 1s, and that position 3 at the right end of the visible tape contains a 0.

**Bounding the size of the problem**

The three sets of logical variables—\( s_{k,t}, p_{i,t}, \) and \( c_{i,t} \)—identified in the preceding section are sufficient to describe the complete history of a Turing machine computation. But how many variables do you need in each set? A Turing machine, after all, can run forever. Moreover, the Turing machine tape extends indefinitely in each direction. Do these facts imply that you need an infinite number of variables to represent the computation history?

Fortunately, the structure of Cook’s theorem makes it possible to limit the number of variables to a finite size. Keep in mind that you are trying to simulate the operation of a nondeterministic Turing machine \( M_Y \) that you know can solve the problem in polynomial time. Thus, you know that the number of steps is bounded by a polynomial \( p_Y \). Because you also know the input value \( Y \), you can determine the maximum number of time steps that the Turing machine will require. Let’s call that number \( T \). Given that the machine will require no more that \( T \) steps for the input value \( Y \), you know that the subscript \( t \) will always fall in the range \( 0 \leq t \leq T \).

The fact that you can bound the number of steps also allows you to limit the set of tape positions. At time \( t \), the Turing machine cannot have moved more than \( t \) squares from its initial position, because the machine moves only one square at a time. Thus, the subscript \( i \) in the variables \( p_{i,t} \) and \( c_{i,t} \) must lie in the range \(-t \leq i \leq t \).

While the total number of variables will grow quite large even for simple problems, it is not infinite. It is, in fact, bounded by a polynomial that depends on the size of the input, which means that this phase of the construction remains within polynomial time.
Constructing the logical expression

Once you have identified the set of variables you need, all that’s left to the proof of Cook’s theorem is finding a way to construct a satisfiability expression using those variables so that the expression is satisfiable if and only if the Turing machine would accept its input. That expression—which must be in conjunctive normal form to meet the conditions of the satisfiability problem—can be divided into the following six sets of conjuncts:

1. *Establish the initial configuration.* The first set of conjuncts guarantees that the machine begins in state 1, the tape head begins in position 0, and the tape contains the input value in the correct position, with 0s in any other square that the machine could read or write during its run. These reachable squares are those that lie no more than $T-1$ squares away from the initial position. In general, this part of the expression has the form

$$s_{1,0} \land p_{0,0} \land c_{(T-1),0} \land \cdots \land c_{1,0} \land c_{0,0} \land c_{1,0} \land \cdots \land c_{T-1,0}$$

The placement of negation bars over the $c$ variables depend on the initial configuration of the tape. Tape squares that contain a 0 correspond to terms that include a negation bar; terms corresponding to tape squares containing a 1 are not negated.

2. *Ensure that the machine is always in exactly one state.* The logical expression you construct must be satisfiable if and only if the Turing machine accepts its input. To make sure that this equivalence holds, the construction must specify every relationship that defines a Turing machine computation, even if those rules seem obvious. Here, for example, you need to guarantee that, for every time step, the machine is in one and only one state. To establish that it must be in at least one state, you can include the conjunct

$$s_{1,t} \lor s_{2,t} \lor \cdots \lor s_{acc,t} \lor s_{rej,t}$$

for each possible time step $t$. To make sure that the machine is not in more than one state simultaneously, you need to include the following conjunct for every time step $t$ and every pair of distinct states $i$ and $j$:

$$\bar{s}_{i,t} \lor \bar{s}_{j,t}$$

This subexpression indicates that, for any pair of state variables at the same time step, at least one of them must be false.

3. *Ensure that the tape head is always in exactly one position.* The logical expressions you need to constrain the tape head to a single position are similar to those used for states in the preceding step. To ensure that the tape head is in at least one position, you need the conjunct

$$\bar{p}_{0,t} \lor \cdots \lor \bar{p}_{1,t} \lor p_{0,t} \lor \bar{p}_{1,t} \lor \cdots \lor \bar{p}_{t,t}$$
To guarantee that the tape head is not in more than one position, you need the following conjunct for every time step $t$ and every pair of distinct positions $i$ and $j$:

$$\bar{p}_{i,t} \lor \bar{p}_{j,t}$$

4. *Restrict tape changes to the position of the tape head.* Turing machines cannot change the contents of the tape except at the current position. The logical expression that represents the Turing machine computation must therefore incorporate this rule. If propositional logic, you can express this rule as follows:

$$((\bar{p}_{i,t} \land \bar{c}_{i,t}) \rightarrow \bar{c}_{i,t+1}) \land ((\bar{p}_{i,t} \land c_{i,t}) \rightarrow c_{i,t+1})$$

In English, the first part of this rule says that if the tape is not in position $i$ at time $t$ and that the tape square at position $i$ contains a 0 at that time, it will still contain a 0 at the time step $t+1$. The second part of the expression makes the same assertion about squares containing 1s. The only problem with this formulation is that the expression is not in conjunctive normal form. Fortunately, you can apply logical identities to rewrite this expression in the following equivalent form:

$$(p_{i,t} \lor c_{i,t} \lor \bar{c}_{i,t+1}) \land (\bar{p}_{i,t} \lor \bar{c}_{i,t} \lor c_{i,t+1})$$

5. *Encode the state transitions of the machine.* The most important part of the construction consists of converting the state table of the Turing machine into an equivalent logical formula. Suppose, for example, that the machine you’re simulating contains the following state:

The instruction in the left-hand box indicates that, if the machine is in state 2 at some time $t$, and the current tape square contains a 0, the situation at time step $t+1$ is that the tape will now have a 1 in that square, the tape head will be one position to the right, and the machine will now be in state 5. This condition can be enforced by including the following expression for every time step $t$ and every reachable position $i$:

$$(s_{2t} \land \bar{c}_{i,t} \land \bar{p}_{i,t}) \rightarrow (c_{i,t+1} \land p_{i+1,t+1} \land s_{5,t+1})$$

As in step 4, you can then apply logical identities to convert the implication into conjunctive normal form, as follows:

$$(s_{2t} \lor c_{i,t} \lor \bar{p}_{i,t} \lor c_{i,t+1}) \land (s_{2t} \lor c_{i,t} \lor \bar{p}_{i,t} \lor p_{i+1,t+1}) \land (s_{2t} \lor c_{i,t} \lor \bar{p}_{i,t} \lor s_{5,t+1})$$
Because the machine is nondeterministic, the entries in the state table may contain multiple instructions. To represent a nondeterministic entry in the satisfiability expression, you need to use an implication that leads to either of the possible outcomes, as illustrated by the following translation of the 1L2, 0R4 entry:

\[(s_{2,l} \land c_{l} \land \bar{p}_{i}) \rightarrow ((c_{l+1} \land \bar{p}_{i+1} \land s_{2,r}) \lor (c_{l+1} \land p_{i+1} \land s_{2,r}))\]

Transforming this expression into conjunctive normal form is a little harder but makes a good test of your ability to manipulate logical formulas.

The one other consideration you need to keep in mind when encoding the state table is that you need to include the accepting and rejecting states. Even though these states never move the tape head or change the tape symbol, it is important to include them in the satisfiability expression to ensure that step 2, which ensures that the machine is always in some state, is not violated. The conjuncts necessary to enforce this rule are constructed in exactly the same form as those for any other state.

6. **Guarantee that the machine ends up in the accepting state.** The final condition that you need to impose on the machine is that it accepts its input. Since the previous step guarantees that the machine will stay in the accepting state if it ever reaches it, all you need to do is make sure that the computation is in the accepting state when the simulation ends at time \(T\). To do so, you need only the single term

\[s_{\text{acc,}T}\]

These six steps generate an expression in conjunctive normal form that is satisfiable if and only if the Turing machine would accept its input.

**An example of the construction**

The construction described in the preceding section is difficult to follow unless you look at a specific example. Suppose that you want to simulate the operation of the \(M_{\text{odd}}\) machine, which was defined earlier in this chapter as follows:

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>reject</td>
<td>1R2</td>
</tr>
<tr>
<td>2</td>
<td>accept</td>
<td>1R1</td>
</tr>
</tbody>
</table>

If you think about the operation of this machine, you’ll quickly see that it makes one transition for each 1 on the input tape, ending up in one of the two final states when it encounters the first 0. If the input represents the number \(N\), the number of steps required—including the transition to the final accepting or rejecting state—is \(N + 1\). Thus, the polynomial that bounds the running time of this machine is simply
\[ p(n) = n + 1 \]

In addition to the machine definition, the construction process depends on a particular input value. To make this process as simple as possible, suppose that all you want to do is simulate the operation of this machine on the following input tape, which represents the number 1:

\[
\cdots 0 \ 1 \ 0 \ \cdots
\]

Given the polynomial bound for the running time, the total number of steps that this machine will run is \( p(1) \), which is 2.

**The final result**

The complete construction of the machine is shown in Figure 10-4. As you can see, even a tiny problem generates a large satisfiability expression. Nonetheless, that formula can be generated mechanically in polynomial time by following the steps enumerated in the preceding section.

The details of the construction used to produce Figure 10-4 are not all that important. In all likelihood, no one will ever in fact go through the details of this construction in practice. What’s important is only that it is possible to do so. That fact means that you could, in theory, solve any problem from the NP class in polynomial time, if only you had a polynomial-time solution to the satisfiability problem. To do so, all you would need to do is carry out these steps:

1. Start with the nondeterministic machine \( M_X \) that solves the particular problem instance \( X \). Such a machine must exist because \( X \) is in NP.
2. Take the instructions for \( M_X \) and the input \( Y \) and use these values to construct an expression in propositional logic that is satisfiable if and only if \( M_X \) accepts the input \( Y \). This construction produces a very large expression, but the length of that expression and the time required to generate it can both be bounded by a polynomial function of the size of the input \( Y \).
3. Determine whether the constructed expression is satisfiable using the polynomial-time solution for satisfiability.

This argument proves that satisfiability is NP-complete.
FIGURE 10-4 Predicate expression generated by applying Cook’s algorithm to $M_{odd}$

Initial configuration

$s_{1,0} \land p_{1,0} \land q_{1,0} \land q_{0,0} \land q_{1,0}$

Ensure that the machine is in exactly one state

$(s_{1,0} \lor s_{2,0} \lor s_{avc,0} \lor s_{avc,0}) \land (s_{1,1} \lor s_{2,1} \lor s_{avc,1} \lor s_{avc,1}) \land (s_{1,2} \lor s_{2,2} \lor s_{avc,2} \lor s_{avc,2}) \land (s_{1,2} \lor s_{2,2} \lor s_{avc,2} \lor s_{avc,2}) \land (s_{avc,0} \lor \overline{s_{avc,0}}) \land (s_{avc,0} \lor \overline{s_{avc,0}}) \land (s_{avc,0} \lor \overline{s_{avc,0}}) \land (s_{avc,0} \lor \overline{s_{avc,0}})$

Ensure that the tape head is in exactly one position

$(p_{1,0} \land p_{1,0} \land p_{1,0}) \land (p_{1,0} \land p_{1,0} \land p_{1,0}) \land (p_{1,0} \land p_{1,0} \land p_{1,0}) \land (p_{1,1} \land p_{1,1} \land p_{1,1}) \land (p_{1,1} \land p_{1,1} \land p_{1,1}) \land (p_{1,1} \land p_{1,1} \land p_{1,1}) \land (p_{1,2} \land p_{1,2} \land p_{1,2}) \land (p_{1,2} \land p_{1,2} \land p_{1,2}) \land (p_{1,2} \land p_{1,2} \land p_{1,2})$

Don’t allow changes to the tape except at the head position

$(p_{1,0} \land q_{1,0} \land q_{1,0}) \land (p_{1,0} \land q_{1,0} \land q_{1,0}) \land (p_{1,0} \land q_{1,0} \land q_{1,0}) \land (p_{1,0} \land q_{1,0} \land q_{1,0}) \land (p_{1,0} \land q_{1,0} \land q_{1,0}) \land (p_{1,0} \land q_{1,0} \land q_{1,0}) \land (p_{1,0} \land q_{1,0} \land q_{1,0}) \land (p_{1,0} \land q_{1,0} \land q_{1,0})$

Transitions for state 1

$(s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{0,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{0,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{0,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{0,0})$

Transitions for state 2

$(s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0}) \land (s_{1,0} \land q_{0,0} \land p_{0,0} \land p_{0,0} \land q_{1,0})$

Rules to remain in the accepting state

$(s_{avc,0} \land q_{avc,0} \land p_{avc,0} \land p_{avc,0} \land q_{avc,0}) \land (s_{avc,0} \land q_{avc,0} \land p_{avc,0} \land p_{avc,0} \land q_{avc,0}) \land (s_{avc,0} \land q_{avc,0} \land p_{avc,0} \land p_{avc,0} \land q_{avc,0})$

Rules to remain in the rejecting state

$(s_{avc,0} \land q_{avc,0} \land p_{avc,0} \land p_{avc,0} \land q_{avc,0}) \land (s_{avc,0} \land q_{avc,0} \land p_{avc,0} \land p_{avc,0} \land q_{avc,0}) \land (s_{avc,0} \land q_{avc,0} \land p_{avc,0} \land p_{avc,0} \land q_{avc,0})$

Rules to accept at the end

$s_{avc,2}$
Computers are most valuable when they are used to solve problems that humans cannot easily solve for themselves. Charles Babbage, for example, wanted to automate the production of mathematical tables, partly because it was a tedious task, but mostly because the people who undertook the necessary calculations made so many mistakes. Computers, however, are also useful when they solve problems faster than human beings. If you face a situation in which timeliness is essential, you may not be able to wait for results generated at human speeds. In such cases, it may be necessary to develop a technological solution to get the answers you need when you need them.

In World War II, the Allies faced precisely this situation. The shipping lanes of the North Atlantic were under such threat from German U-boats that Britain was in danger of being starved into submission. Breaking the U-boat code was a critical turning point in the war and may have changed its outcome. Faced with a code that changed every day, the British had to develop mechanical tools that would allow them to read German military dispatches quickly enough to act on that information.

Breaking the German military codes was an early application of cryptography, which is the science of creating and decoding messages whose meaning cannot be understood by those who intercept the message. In the language of cryptography, the message you are trying to send is called the plaintext; the message you actually send is called the ciphertext. Unless your adversaries know the secret of the encoding system, which is usually embodied in some privileged piece of information called a key, intercepting the ciphertext should not make it possible for them to discover the original plaintext version of the message. On the other hand, the recipient, who is presumably in possession of the key, can easily translate the ciphertext back into its plaintext counterpart.
The Navajo code talkers

As you will discover in this chapter, cryptography was one of the earliest applications of modern computing. During World War II, a codebreaking team in England, building on earlier work carried out in Poland, developed specialized hardware that was able to break the German Enigma code. Breaking that code was critical to the Allied victory in the battle for control of the Atlantic shipping lanes.

World War II offers other cryptographic stories as well—stories that underscore the fact that high technology does not necessarily offer the best solution to the problem of secure communication. In the war against Japan, the United States Marine Corps relied on the Navajo, a Native American tribe from the southwestern United States, to exchange messages over radio channels on which anyone might be listening. Approximately 400 Navajos served as “code talkers” from 1942 to 1945 and played a vital role in the war effort.

Howard Connor, signal officer for the 5th Marine Division observed that “were it not for the Navajos, the Marines would never have taken Iwo Jima.”

The code talkers did not simply speak Navajo over the radio. Military messages often include words that do not exist in Navajo, along with place names and other words that are hard to translate. If, for example, you wanted to send a message warning of submarines off Bataan, you would have to decide how to express submarine and Bataan, neither of which has a Navajo counterpart.

To solve this problem, the code talkers used a variety of strategies. For common military terms, Navajo words were used to provide an appropriate metaphor; submarine, for example, was expressed using the Navajo words for iron fish. Place names were translated using a spelling strategy involving both English and Navajo. To send the word Bataan, for example, the code talkers first spelled it out using English words beginning with the appropriate letters. One possibility looks like this:

bear apple tooth axe ant needle

The code talker would then substitute the Navajo words and deliver the following message:

shush be-la-sana a-woh tse-nill wol-la-chee tsah

The native speaker on the receiving end would listen for each word, translate it back from Navajo to English, and then record the initial letters.

It is important to note that the spelling scheme used by the code talkers allows many words to stand for the same letter. The three occurrences of the letter a in Bataan are each represented by a different Navajo word, making the code much more difficult to break.

The Navajo code talkers proved to be much faster than the encryption strategies adopted by the other service branches. A well-trained pair of code talkers could transmit a three-line message in 20 seconds; the fastest encryption machines of the day required 30 minutes to deliver the same message. More importantly, the code-talker strategy proved to be more secure. The Japanese were able to break the codes used by the Army and Army Air Core, but were never able to decipher the messages sent by the Navajo code talkers.

On September 17, 1992, the surviving members of the Navajo code talkers were honored at the dedication of a commemorative exhibit at the Pentagon in Washington, DC.
11.1 Early history of cryptography

Cryptography has been around in some form or another for most of recorded history. There is evidence to suggest that coded messages were used in ancient Egypt, China, and India, possibly as early as the third millennium BCE, although few details of the cryptographic systems have survived. In Book 6 of the *Iliad*, Homer suggests the existence of a coded message when King Proitos, seeking to have the young Bellerophontes killed, has

\[\ldots\] sent him to Lykia, and handed him murderous symbols, which he inscribed on a folding tablet, enough to destroy life.

Hamlet, of course, has Rosencrantz and Guildenstern carry a similarly dangerous missive, but Hamlet’s message is secured under a royal seal. In the *Iliad*, there is nothing to suggest that Bellerophontes cannot see the “murderous symbols,” which implies that their meaning must somehow be disguised.

One of the first encryption systems whose details survive is the Polybius square, developed by the Greek historian Polybius in the second century BCE. In this system, the letters of the alphabet are arranged to form a 5×5 grid in which each letter is represented by its row and column number.

Suppose, for example, that you want to transmit following English version of Pheidippides’ message to Sparta:

**THE ATHENIANS BESEECH YOU TO HASTEN TO THEIR AID**

This message can be transmitted as a series of numeric pairs, as follows:

\[
\begin{align*}
44 &\quad 23 &\quad 15 &\quad 11 &\quad 44 &\quad 23 &\quad 15 &\quad 33 &\quad 24 &\quad 11 &\quad 33 &\quad 43 &\quad 12 &\quad 15 &\quad 43 &\quad 15 &\quad 13 &\quad 23 &\quad 54 \\
34 &\quad 45 &\quad 44 &\quad 34 &\quad 23 &\quad 11 &\quad 43 &\quad 15 &\quad 33 &\quad 44 &\quad 34 &\quad 44 &\quad 23 &\quad 15 &\quad 24 &\quad 11 &\quad 24 &\quad 13
\end{align*}
\]

The advantage of the Polybius square is not so much that it allows for secret messages, but that it simplifies the problem of transmission. Each letter in the message can be represented by holding between one and five torches in each hand, which allows a message to be passed quickly over great distances. By reducing the alphabet to an easily transmittable code, the Polybius square anticipates such later developments as Morse code and semaphore, not to mention modern digital encodings such as ASCII or Unicode.

In *De Vita Caesarum*, written sometime around 110 CE, the Roman historian Suetonius describes an encryption system used by Julius Caesar, as follows:

If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.
Even today, the technique of encoding a message by shifting letters a certain distance in the alphabet is called a **Caesar cipher**. According to the passage from Suetonius, each letter is shifted three positions ahead in the alphabet. For example, if Caesar had had time to translate his final words according to his coding system, 

**ET TU BRUTE** would have come out as **HW WX EUXWH**, because **E** gets moved three letters ahead to **H**, **T** gets moved three to **W**, and so on. Letters that get advanced past the end of the alphabet wrap around back to the beginning, so that **X** would become **A**, **Y** would become **B**, and **Z** would become **C**.

Caesar ciphers have survived into modern times. On the early electronic bulletin boards that were popular at the beginning of the Internet era, users could disguise the content of postings that might offend some readers by employing a mode called **ROT13**, which is simply a Caesar cipher that shifts all letters forward 13 positions. And the fact that **HAL**—the name of the computer in Arthur C. Clarke’s *2001*—is a one-step Caesar cipher of IBM has generated some amount of interest among fans.

Although Caesar ciphers are certainly simple, they are also extremely easy to break. There are, after all, only 25 possible Caesar ciphers for English text. If you want to break a Caesar cipher, all you have to do is try each of the 25 possibilities and see which one translates the ciphertext message into something readable.

A somewhat more secure scheme is to allow each letter in the plaintext message to be represented by some other letter, but not one that is simply a fixed distance from the original. In this case, the key for the encoding operation is a letter translation table that shows what each of the possible plaintext characters becomes in the ciphertext. Such a coding scheme is called a **letter-substitution cipher**.

Letter-substitution ciphers have been used for many, many years. Examples of such ciphers appear in several works from both classical and medieval times. In the early 15th century, the Arabic encyclopedia *Subh al-a ’sha* included a section on cryptography describing various methods for creating ciphers as well as techniques for breaking them. In particular, this manuscript included the first instance of a cipher in which several different coded symbols can stand for the same plaintext character. Codes in which each plaintext letter maps into a single ciphertext equivalent are called **monoalphabetic ciphers**; codes in which each character can have more than one coded representation are called **polyalphabetic ciphers**.

### 11.2 Cryptograms

Today, monoalphabetic ciphers survive primarily in the form of letter-substitution puzzles called **cryptograms**. Edgar Allan Poe was a great fan of cryptograms and included a cryptographic puzzle in the excerpt from *The Gold Bug* shown in Figure 11-1.
Here Legrand, having re-heated the parchment, submitted it to my inspection. The following characters were rudely traced, in a red tint, between the death's head and the goat:

53††;305)6;482644644;806;48†8[60]85;1†(683885†;46(;88+96+7;8)*†(485);5†2:*†(49562(564889888*;4069285);6†8)4††;1(t9;48081;8;8†1;48†85;4)485†528806814†9;48;884(‡34;48)4‡;161;:188;‡;

“But,” said I, returning him the slip, “I am as much in the dark as ever. Were all the jewels of Golconda awaiting me upon my solution of this enigma, I am quite sure that I should be unable to earn them.”

“And yet,” said Legrand, “the solution is by no means so difficult as you might be led to imagine from the first hasty inspection of the characters. These characters, as any one might readily guess, form a cipher … such, however, as would appear to the crude intellect of the sailor, absolutely insoluble without the key.”

“And you really solved it?”

“Readily; I have solved others of an abstruseness ten thousand times greater. Circumstances, and a certain bias of mind, have led me to take interest in such riddles, and it may well be doubted whether human ingenuity can construct an enigma of the kind which human ingenuity may not, by proper application, resolve. In fact, having once established connected and legible characters, I scarcely gave a thought to the mere difficulty of determining their import.

“My first step was to ascertain the predominant letters, as well as the least frequent. Counting all, I constructed a table thus:

<table>
<thead>
<tr>
<th>Of the character</th>
<th>There are</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>?</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>-•</td>
<td>1</td>
</tr>
</tbody>
</table>

“Now, in English, the letter which most frequently occurs is e. Afterward, the succession runs thus:

ao id h n r s t u y c f g l m w b k p q x z

“. . . Let us assume g, then, as e. Now, of all words in the language, the is most usual; let us see, therefore, whether there are not repetitions of any three characters, in the same order of collocation, the last of them being g. If we discover a repetition of such letters, so arranged, they will most probably represent the word the. Upon inspection, we find no less than seven such arrangements, the characters being ;48. We may, therefore, assume that ; represents t, 4 represents h, and 8 represents e—the last being now well confirmed. . . .

“But, having established a single word, we are enabled to establish a vastly important point; that is to say, several commencements and terminations of other words. Let us refer, for example, to the last instance but one, in which the combination ;48 occurs—not far from the end of the cipher. We know that the ; immediately ensuing is the commencement of a word, and, of the six characters succeeding this th, we are cognizant of no less than five. Let us set these characters down, thus, by the letters we know them to represent, leaving a space for the unknown—t_e_e_e

“Here we are enabled, at once, to discard the th as forming no portion of the word commencing with the first t; since, by experiment of the entire alphabet for a letter adapted to the vacancy, we perceive that no word can be formed of which this th can be a part. We are thus narrowed into t_e_e, and, going through the alphabet, if necessary, as before, we arrive at the word tree as the sole possible reading. We thus gain another letter, r . . . .

“I have said enough to convince you that ciphers of this nature are readily soluble, and to give you some insight into the rationale of their development . . . . It now only remains to give you the full translation of the characters upon the parchment, as unriddled. Here it is:

A good glass in the bishop's hostel in the devil's seat forty-one degrees and thirteen minutes northeast and by north main branch seventh limb east side shoot from the left eye of the death's head a bee-line from the tree through the shot fifty feet out.
(a) The following coded message is an enciphered version of the opening paragraph from a well-known English novel after removing all spaces and punctuation and then breaking the message into five-letter groups:

LESTX KQLEY TQOJX ZEHYT QJQKL IQHST XAALY EXYSE SRYPH LJYPG
QYTXX QVLKK QHGLY TYTQO EHRKV GXXFR SXFPR BQQKE XJPQY
SHJPA SJQRS BHPTX KQGLY TEXYT LEOLE LYYXR LHYXG EXEXJ YXQSY
LYGSR STXAA LYTXK QSEHY TSYBQ SERDX BVXJY

Use Poe’s strategy to decipher this message. Remember that the letter frequencies are just an approximation and that E is not always the most common letter.

(b) In the Sherlock Holmes mystery, *The Adventure of the Dancing Men*, by Sir Arthur Conan Doyle, Holmes receives several messages written in what appears to be “a number of absurd little figures dancing across the page upon which they are drawn.” See if you can apply Poe’s techniques to this cipher, which did not stump Holmes for long:

Message 1:

Message 2:

Message 3:

Message 4:

Message 5:

Message 6:
In describing the solution to Captain Kidd’s message, Poe offers a general technique for solving monoalphabetic ciphers: calculate the frequency of the letters used in the ciphertext and correlate the appearance of coded sequences with the frequency of letters in English. By guessing that the letters appearing most often in the ciphertext correspond to the most common letters in English, you can usually make a good start toward solving such puzzles.

If you try to solve cryptograms on your own, however, it will help you to know that Poe’s list of the most common letters is not in fact correct. Computerized analysis reveals that the most common letters in English are

E T A O I N S H R D L U

Given that statistical studies of English text were by no means as well developed in Poe’s day, Poe can perhaps be excused for making a few mistakes.

What Poe did realize is that solving a monoalphabetic cipher requires a strategy. The Caesar cipher, for example, requires one to check only 25 possibilities before the correct plaintext must appear. In the general case of a letter-substitution cipher, there are 26 possible letters to choose as the coded representation for A, 25 remaining possible letters to choose as the coded representation for B, 24 possibilities for C, and so on, for a total of $26! = 26 	imes 25 	imes 24 \times \ldots \times 3 \times 2 \times 1$ possible encodings. This number is extremely large, equal in decimal notation to 403,291,461,126,605,635,584,000,000. Even with modern computers, it isn’t feasible to solve this problem by trying every possibility. One needs instead to be more subtle.

11.3 The Enigma machine

In many ways, modern computing got its start during World War II. On both sides, the war focused attention on military priorities and made it possible to apply unprecedented levels of resources in an attempt to gain the advantage. The Germans, for example, made enormous investments in missile technology, which led to the development of the V-1 and V-2 rockets that fell with such devastating effect on England during the Blitz. In the United States, the Manhattan Project brought together the leading scientists of the day to develop the atomic bomb.

As noted in the introduction to this chapter, the war forced Britain to apply considerable resources to the problem of deciphering messages that the German High Command used to communicate with the army, navy, and air force. Although each service branch used a slightly different technology, all were built upon a common foundation that made it possible for the Allies to break those codes.
In the early 1930s, the German military adopted a new encryption protocol based on an existing commercial device called Enigma. Figure 11-2 shows the top view of a typical Enigma machine, expanded so that you can see the detail. At the bottom of the figure is a keyboard arranged in the standard German layout. Above the keyboard is an array of lamps. Pressing a key lights one of the lamps, thereby indicating the encoded version of that letter. The mapping from keys to lamps is controlled by the three thumb wheels at the top of the diagram, which are called rotors. Each rotor can be set to any of 26 positions corresponding to the letters of the alphabet. The display windows at the top of Figure 11-2 show the letters JLY, which is called the rotor setting.

Early models of the Enigma machine included only the components shown in Figure 11-2. These machines had 17,576 (26 × 26 × 26) settings, which made it possible, given sufficient time, to decrypt a message by trying every rotor setting.
To increase the security of Enigma, the German government mandated several changes in its design. Instead of using a fixed set of rotors, the Enigma machines used during the war allowed operators to select and arrange any three rotors from a set of five. This change meant that codebreakers had to consider $60 (5 \times 4 \times 3)$ possible rotor arrangements. Military models of the Enigma machine added a ring inside each rotor that introduced an additional offset into the transformation and changed the point at which the next rotor advanced. The addition of the ring had no real impact on the decryption strategy and is not considered in this chapter.

From the perspective of would-be codebreakers, the change that added the most complexity was the introduction of a new front panel containing jacks associated with the letters of the alphabet, as shown in Figure 11-3. In German, this panel was called the *steckerbrett*, which is traditionally rendered in English as *steckerboard*. Enigma operators were issued a set of cables that allowed them to exchange pairs of letters during the encryption. Although it’s hard to follow the tangle of cables in the photograph, the steckerboard wiring in Figure 11-3 exchanges the pairs of letters A-D, B-X, I-Z, J-U, and L-R. Letters connected in this way are called *stecker pairs*.

The addition of the steckerboard vastly increased the number of possible settings for the Enigma machine. The set of ten plug wires Enigma operators were issued during the war allowed for $216,751,064,975,576$ possible wirings. Taken together with the $17,576$ possible rotor settings and the $60$ possible ways to select and arrange the rotors, the number of initial settings of the Enigma machine was the astronomical $228,577,003,080,643,426,560$. Even with today’s technology, trying every possible combination would take a considerable amount of time. Given the technology available in World War II, trying every possibility was not a realistic option. Decoders had to rely on cleverness and insight—along with a bit of mechanical assistance—to break the Enigma code.

![Figure 11-3 The Enigma steckerboard](image-url)
11.4 The codebreakers

In 1938, recognizing the danger of war in Europe, the head of British intelligence purchased an estate about 50 miles northwest of London called Bletchley Park, which became the home of the Government Code and Cipher School. More than 10,000 people worked at Bletchley Park during the war, under the strictest secrecy. The task of breaking Enigma fell to a team of cryptographers at Bletchley Park working under the code name Ultra. The Ultra team employed many of Britain’s best mathematicians, including Alan Turing, the inventor of the Turing machine described in Chapter 8. Despite its enormous complexity, the mathematicians of Ultra managed to break the Enigma code. In fact, they did so several times.

Cryptography is in many ways a race between codemaker and codebreaker. The Germans made periodic improvements to the Enigma both before and during the war. With each redesign, the codebreakers had to come up with a new strategy to overcome the enhancements on the German side. When the German navy added a fourth rotor to the Enigma in February 1942, the Allies were unable to read Enigma traffic for ten months. By the end of the war, however, Bletchley Park was able to decipher most encrypted messages in less than a day.

Being able to read German military communications was vital to the Allied cause. In 1941, Alan Turing and several of his colleagues wrote directly to Prime Minister Winston Churchill requesting more resources for the decryption effort. Fully aware of the importance of the Ultra project, Churchill replied

Make sure they have all they want on extreme priority and report to me that this had been done. Action this day.

After the war, Churchill is reported to have told King George VI that “it was thanks to Ultra that we won the war.”

The cryptographers at Bletchley owed a considerable debt to the Polish cryptographers Marian Rejewski, Jerzy Różycki, and Henryk Zygalski, who were able to break the Enigma code in 1932. In the process, they also developed many of the cryptographic techniques that would later guide the British effort. Fortunately, the Polish team was able to share its decryption work with the Allies shortly before the German invasion of Poland in 1939 that marked the beginning of the war. The Polish team later made their way to France, where they carried on their cryptographic work along with French colleagues. When France itself was overrun, the Poles again escaped to England. Although the secrecy around the wartime cryptographic work meant that the Polish contribution to codebreaking remained unknown for many years, Bletchley Park now has a monument to commemorate the essential work of these Polish mathematicians.
11.5 The internal structure of Enigma

Before you can understand how cryptographers were able to break the Enigma code, you need to know something about how the machine works. Figure 11-4 shows the internal structure, focusing on the wiring of the rotors and the steckerboard.

Each of the three rotors in the Enigma machine has 26 contacts along its left and right sides. Current that comes in at one contact on the rotor is redirected to a contact on the opposite side according to the internal wiring pattern, which is different for each rotor. Each rotor therefore implements a reordering of the letters, which mathematicians call a permutation. The steckerboard also implements a permutation, which is set manually according to the instructions in codebook.

The letters at the top of Figure 11-4 indicate the rotor setting. Typing a character on the keyboard automatically advances the rotor on the right, thereby changing the pattern of connections inside the machine. When that rotor has completed a full revolution, the middle rotor advances one step; in much the same way, completing a revolution of the middle rotor advances the rotor on the left. The rotors therefore advance in a fashion reminiscent of the odometer on a car. The right rotor advances...
on every character and is therefore called the fast rotor. The middle rotor advances once every 26 characters and is called the medium rotor. The left rotor advances only once every 676 (26 × 26) characters and is unsurprisingly called the slow rotor.

Figure 11-5 shows what happens if the operator types the letter A on the keyboard. Pressing the key advances the fast rotor, which changes the rotor setting from JLY to JLZ. The Enigma machine then applies a current to the wire leading from the A key at the right edge of the diagram and, at the same time, disconnects the A lamp so that only the encrypted version of the letter appears. The current flows across the steckerboard, then through the three rotors from right to left. It then passes into a circuit element called the reflector, which implements a fixed permutation. From the reflector, the current flows back across the rotors in the opposite direction and then passes through the steckerboard one more time. As shown in the diagram, the current initiated by typing A ends up on the wire labeled K, which causes the K lamp to light. Thus, given the rotor setting JLZ, the ciphertext form of the letter A is K.

The encryption patterns generated by the Enigma machine are difficult to break because the machine implements a polyalphabetic cipher in which the encoding
changes on every character. If, for example, the operator types a second letter A immediately after the first, the machine advances to the configuration shown in Figure 11-6. This time, the fast rotor and the medium rotor both advance, because the fast rotor has made it all the way through to the end of the alphabet. Given the rotor setting JKA that appears after both rotors have moved forward, the letter A is now translated into the letter Q.

At this point, it is useful to note a fundamental symmetry in the Enigma design. If A is transformed to Q at some rotor setting, it must also be the case that Q is transformed to A. The circuit is exactly the same; the only difference is that the current flows in the opposite direction. This symmetry is very useful for Enigma operators because it means that the sender and receiver don’t need to have two different keys. The sender sets the rotors and the steckerboard according to a codebook and types in the message. What comes out in the lights is the ciphertext, which is typically transmitted over a radio channel in Morse code. As long as the receiver uses the same codebook and sets up the machine in the same way, typing in the ciphertext restores the original message, because the encryption is reversible. As you will discover in the next section, however, the fact that the Enigma encoding is reversible also makes life easier for anyone trying to break the Enigma code.
11.6 Breaking the Enigma code

The decryption strategy developed in Poland and refined at Bletchley Park made use of the following facts about the Enigma machine:

- *The Enigma encoding is symmetrical.* As noted in the preceding section, if the A key is transformed into the letter Q, it must be the case that the Q key would be transformed into A for that particular rotor setting.

- *The Enigma machine can never map a character into itself.* Because of its construction and the symmetry of the transformation, it is never possible to have the letter A, for example, come back as the letter A.

- *The steckerboard does not affect the transformation pattern of the rotors, but only the characters to which the outputs of that rotor are assigned.* Although the addition of the steckerboard vastly increases the number of possible encodings, it does not change several fundamental properties of the machine.

The codebreakers were also fortunate that the German military was rigid in its communication style, which made it possible to anticipate what the content of a message might be. In particular, the Germans routinely transmitted weather reports at specific times of the day, which were often straightforward to guess if you knew what the weather looked like at the point of transmission. Similarly, many messages tended to start with a salutation to the receiving general, admiral, or captain in a way that included the full name and title. In salutations, the German word for to is an, which meant that the first characters in an intercepted message might be ANGENERAL (or ANXGENERAL for those branches of the German military that used the letter X to indicate a space). In fact, it was sometimes sufficient to guess that the first three characters in a message were ANX without having any idea of who the intended recipient might be.

The known-plaintext attack

The strategy of breaking a code by guessing at least part of the plaintext and then using that guess to deduce the encryption pattern is called a *known-plaintext attack.*

The character sequence that you believe you know is called a *crib.* Ironically, one of the best cribs available to Project Ultra—at least according to some accounts—occurred in messages from a German officer in the North Africa campaign who foolishly sent periodic messages containing the German equivalent of nothing to report, which is keine besonderen ereignisse.

Suppose that one of the Allied listening posts in North Africa had intercepted the following coded message:

UAUNFYRLPZSWMEDSINFKRJXFSXKJCAXKEZ
If the sender is behaving in his usual way, you suspect that this message contains the plaintext sequence

**KEINEBESONDERENEREIGNISSE**

If you can figure out where in the message this sequence occurs, you might then be able to use the pattern of letters to make deductions about the settings of the Enigma machine. If these deductions allow you to determine the rotor pattern and the wiring of the steckerboard, you have broken the Enigma code for that day.

**Aligning the crib with the ciphertext**

The first challenge in implementing the known-plaintext attack consists of figuring out where in the ciphertext the suspected crib might occur. Fortunately, many of the potential positions for the crib can be ruled out simply by taking note of the fact that the Enigma machine never translates a letter to itself. For example, the crib cannot occur at the beginning of the ciphertext because the letter *N* would have to map to itself in the fourth character position, as would the letter *E* a bit further on, as shown in the following diagram:

```
U A E N F V R L B Z P W M E P M I H F S R J X F M J K W R A X Q E Z
KEINEBESONDERENEREIGNISSE
```

The codebreakers at Bletchley used the word *crash* to refer to positions at which a letter in the ciphertext matches its counterpart in the crib. The first step in the decryption process is to slide the crib under the ciphertext until no crashes occur.

Figure 11-7 on the next page shows what happens if you carry out this process for every possible alignment of the crib and ciphertext. There are only two possible alignments that produce no crashes, which arise from shifting the crib five and six characters to the right, respectively. If the crib is correct, it must be in one of those two positions.

After eliminating the alignments ruled out because of crashes, the cryptographers at Bletchley would then try each of the possible alignments to see whether any of the remaining possibilities gave rise to a consistent rotor setting.

**Deducing the rotor setting**

Once you have a possible alignment, you can use the patterns of letters in the crib and the ciphertext to make inferences about the rotor setting. The basic idea is that only certain settings of the rotors will produce the pairings of letters you see between the crib and the appropriate region of the ciphertext. If you could use that information to eliminate all but a few of the possibilities, you could then check those settings by hand.
Crashes that rule out certain alignments between the crib and the ciphertext

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]

\[\text{UAE} \text{NFVRLBZPWM}E \text{PMIHF} \text{SRJXFMJKWRA} \text{XQEZEKINEBE} \text{SOND} \text{ERENEREIGNISSE} \]
If there were no steckerboard, this process would be entirely straightforward. What you are looking for is a rotor setting that transforms some portion of the ciphertext back into the crib. Suppose, for example, that you assume that the crib appears at an offset of 5, as shown in the first boxed possibility in Figure 11-7. What you then need to do is find some setting of the rotors at which typing in

\[ \text{YRLPZSWMEDSINFKRJXFSXKJCA} \]

gives you back

\[ \text{KEINEBESONDEREREIGNISSE} \]

Carrying out this analysis manually would certainly be time-consuming, but there are only 1,054,560 possible arrangements and settings for the rotors. If all 10,000 people at Bletchley Park—working in parallel—were able to test one of these settings every minute, you would find the solution in less than two hours.

Of course, given the resources available to Bletchley Park under Churchill’s designation of “extreme priority,” it would not have been necessary to divert all of Bletchley’s personnel to test the configurations. Given the technology of the time, it was possible to build a mechanical device to step through the 1,054,560 arrangements and settings of the rotors, checking for a match.

Unfortunately, the existence of the steckerboard rules out this simple strategy. Even if you find the right rotor settings, typing in

\[ \text{YRLPZSWMEDSINFKRJXFSXKJCA} \]

won’t regenerate the crib, because the letters are transformed by the connections on the steckerboard. If testing all possible arrangements of the rotors takes two hours, adding in the complexity of trying all 216,751,064,975,576 steckerboard wirings means that the process would take on the order of 10 billion years, which is a rough approximation of the age of the universe.

The critical insight that allowed the allies to break Enigma is that certain patterns in the letter pairings between the crib and the ciphertext are independent of the steckerboard. Consider, for example, the circled pairs of letters in the presumed alignment at offset 5:

\[ \begin{align*}
\text{VRLBZPWMEPMIHFSRJXFJMJKWRA} \\
\text{KEINEBESONDEREREIGNISSE}
\end{align*} \]

The numbers below the characters keep track of the index of the character in the crib, beginning—as is conventional in computer science—at index position 0.
Assuming that the crib and offset are correct, the Enigma machine encodes the plaintext \textbf{N} into the ciphertext \textbf{B} at index 3. Two characters later at index 5, the machine turns \textbf{B} into the ciphertext \textbf{P}. At index 9, the letter \textbf{N} becomes a \textbf{P}. Given the symmetry of the Enigma machine, however, you know that typing a \textbf{P} at index 9 would have produced an \textbf{N}, which is the letter that began this chain back at offset 0. The transformation pattern of \textbf{N} to \textbf{B}, \textbf{B} to \textbf{P}, and \textbf{P} to \textbf{N} form a closed cycle, which is easier to see if you connect the matching letters like this:

\[
\begin{array}{cccccccccccccccc}
K & E & I & N & E & B & S & O & N & D & E & R & E & N & E & I & G & N & I & S & S & E \\
\end{array}
\]

Alan Turing used the term \textit{loop} to refer to this sort of closed cycle in the letter pairings between the crib and the ciphertext. The wonderful property of loops is that they are unaffected by the configuration of the steckerboard. Different settings of the steckerboard generate different letters in the ciphertext, but a cycle that occurs with one steckerboard setting will also occur if that setting is changed.

The easiest way to find the loops in some alignment between the crib and the ciphertext is to construct what Turing called a \textit{menu}, which is simply a diagram showing the connections that appear between the letters. In the current example, index position 0 links \textbf{K} to \textbf{V}. The menu therefore contains the following pairing:

\[
\begin{array}{c}
K \quad 0 \quad V \\
\end{array}
\]

The complete menu for this crib-ciphertext alignment appears in Figure 11-8.
Once you have completed the diagram, the loops jump out visually. In the menu shown in Figure 11-8, there is one loop of length 2 (E→R→E), two loops of length 3 (F→I→E→F and N→B→P→N), two loops of length 4 (E→R→S→W→E and R→J→N→S→R), and one loop of length 6 (E→R→J→N→S→W→E).

The discovery of these loops gave the Bletchley team the breakthrough they needed to crack the Enigma code. The fact that the loop pattern is independent of the steckerboard means that you can deduce the rotor patterns simply by running through all the possible settings. To speed up that process, the Bletchley team built an electromechanical computing device called the **Bombe**, which simulated the operation of the Enigma machine. The Bombe was programmed to search for feasible rotor positions given a particular set of loops in the encoding of a suspected plaintext into its encrypted version. At each state of the machine’s operation, it would assume that the current setting of the rotors was correct. If that assumption led to a contradiction, the Bombe would quickly move on to the next cycle. Although the running time depended on the number of loops detected in the crib-ciphertext pairing, the Bombe was typically able to search through all possible rotor combinations in less than an hour.

Breaking the code for one intercepted message did not give Bletchley Park the ability to read all the Enigma transmissions for that day. The Germans were clever enough to realize that it would be foolish to transmit a large number of messages with the same encryption key. What they did instead was to have the Enigma operator come up with his own encryption key and then encipher that key—using the settings from the codebook—before sending the actual message. For example, if the rotor setting for the day was **JLY**, the operator would initialize the machine to that setting and then transmit a new message key of his own devising. The operator would then reset the machine so that it used the new key to encode the rest of the message. The receiver would simply reverse the process. After setting the machine to the settings from the codebook, the receiver would then use the characters from the beginning of the message to reset the machine appropriately.

All too often, these operator-chosen keys were too easy to predict. Lazy operators might choose a key that was easy to type **AAA**. Others might use names of friends and family such as **PIA**. If the Bletchley codebreakers could guess the
message key, the decryption operation became much easier. More damaging to German security, however, was the fact that message keys—at least in the early days of the Enigma—were transmitted twice in succession to make sure they got through. This procedure left an enormous hole in the German encryption strategy. If codebreakers knew that the first and fourth, second and fifth, and third and sixth letters in a message always represented the same plaintext letter, they could use this knowledge to guess the rotor settings. The Polish decryption strategy used this method and therefore did not rely on being able to find a crib.

Breaking a single message, however, represented a real victory because doing so typically allowed the codebreakers to determine the setting of the steckerboard. After setting up an Enigma machine so that it matched the setting of the message key, encoding the appropriate section of the ciphertext would yield a string of characters that was a simple letter-substitution cipher of the crib, which is generally easy to solve. Knowing both the rotor order and the steckerboard wiring—neither of which change from message to message over the course of a single day—made it easy to decode other messages for that day because there were only the 17,576 rotor settings to check.

As Winston Churchill’s report to King George makes clear, the cryptographic work at Bletchley Park helped the Allied cause enormously. Cryptography is still important as a field of study today. In the 1970s, computer scientists developed an entirely new to encryption called public-key cryptography that has revolutionized electronic communication, which you will have the chance to explore in Chapter 12.
In his discussion of codebreaking in the excerpt from *The Gold Bug* in Figure 12-1, Edgar Allan Poe—in the persona of detective Legrand—expresses his doubt that “human ingenuity can construct an enigma of the kind which human ingenuity may not, by proper application, resolve.” Although Poe’s speculation suggests that codemakers are forever at the mercy of codebreakers, recent discoveries have made the prospect of a truly unbreakable code much more likely. Much of that change comes from a radical new approach to cryptography that has transformed the process of encrypting data. That approach is called public-key *cryptography*, which, as its name implies, allows encryption keys to be distributed publicly without compromising the security of the communication channel.

The discovery of public-key cryptography has had enormous practical impact. Lawrence Lessig, one of the leading experts on law and technology, went so far as to make the following claim in his 1999 book *Code and Other Laws of Cyberspace*:

> Here is something that will sound very extreme but is at most, I think, a slight exaggeration: encryption technologies are the most important technological breakthrough in the last one thousand years. No other technological discovery . . . will have a more significant impact on social and political life. Cryptography will change everything.

By eliminating the need for any prearrangement between communicating parties, public-key cryptography makes it possible to include security and authentication in a wide variety of applications. Public-key encryption makes it possible to send information securely through the World-Wide Web and is the enabling technology behind the new generation of credit cards that contain computer chips—the so-called “smart” cards. Encryption is central to our communications age, which could not exist without the security provided by public-key cryptosystems.
Should cryptography be secret

The fundamental purpose of cryptography is to keep information secret. That principle obviously applies to the content of encrypted messages, but what about the science of cryptography itself? Do governments have a legitimate interest in keeping cryptography secret or is it more important to secure the public benefits of releasing cryptographic applications to the world? That question has generated considerable debate in recent decades and has had a profound influence on the development of computer science.

During the Second World War, maintaining strict secrecy about the decoding work at Bletchley Park was vital to the Allied cause. If the Germans had any inkling that the British had broken their Enigma code, they would have changed their encryption strategy, which in turn might have altered the course of history. After the war, the British intelligence services maintained their wartime policy of absolute secrecy. After the war, the Government Code and Cipher School was renamed the Government Communications Headquarters (GCHQ), but work on cryptography remained strictly controlled.

In the United States, research on cryptography followed a different path. Although the intelligence agencies such as the National Security Agency (NSA) continued to work on cryptography, academic research in this area moved forward as well.

The strategy of public-key encryption described in this chapter grew out of academic research, which—in contrast to the intelligence community—maintains a tradition of free and open communication. As a result, the ideas of public-key encryption circulated widely and were picked up, not only by scientists and academics, but also by companies that used public-key encryption to transform the very nature of electronic communication.

The security establishment in the United States, however, was uncomfortable with the release of cryptographic secrets and therefore sought to limit distribution of cryptographic algorithms, particularly to foreign nationals. For twenty years, strong cryptographic techniques, including the public-key systems described in this chapter, were classified as “munitions” and restricted for export under the International Traffic in Arms Regulations (ITAR). Those restrictions remained in place until 1996.

In recent years, the drive to keep cryptography under wraps has taken a new direction as encryption becomes central to the protection of intellectual property. The 1998 Digital Millennium Copyright Act makes it illegal to “circumvent a technological measure” designed to restrict access to copyrighted works. Although the full impact of the law is still unclear, threats of prosecution have already forced researchers to withdraw papers describing flaws in commercial encryption schemes.

Ironically, the post-war obsession with secrecy led Britain to miss one of the most important discoveries in modern computer science. Working at GCHQ in the 1970s, James Ellis, Clifford Cocks, and Malcolm Williamson discovered the principles of public-key cryptography and—a few years after Ellis’s original paper—managed to devise a practical implementation that is essentially identical to the RSA algorithm described in this chapter. Unfortunately, their work was completely unknown until it was declassified in 1997.

---

THE POSSIBILITY OF SECURE NON-SECRET DIGITAL ENCRYPTION

J. H. Ellis, January 1970

Introduction

1. It is generally regarded as self-evident, that, in order to prevent an interceptor from understanding a message which is intelligible to the authorised recipient, it is necessary to have some initial information known to the sender and to the recipient but kept secret from the interceptor. This information can take many forms, such as the method of encipherment itself, the construction of a cipher machine, a key setting or a one-time tape. All these methods require that there is a route by which this secret information can be sent without fear of interception. Only then can the cipher text be sent safely in a non-secret manner, and large quantities of cipher text of high security thus tend to need the parallel transmission of a tape, but still substantial quantities of secret information.

2. This report demonstrates that a secret information is not theoretically necessary and that, in principle, secure message can be sent even though the method of encipherment and all transmissions between the authorised communicators are known to the interceptor. This is what is meant by “non-secret encryption”. It must be emphasised however that this demonstration has only the status of an existence theorem. It shows only that such a system is theoretically possible, and not that a practical form exists. The demonstration consists of showing that a particular, but unfortunately as yet highly impractical, system has the desired properties. This is followed by an heuristic discussion which attempts to establish the necessary properties of a system and indicate the likely form of a practical solution.

Introduction to James Ellis’s memo on non-secret encryption
Public-key cryptography was originally discovered in 1970 by James Ellis, a British scientist at the Government Communications Headquarters (GCHQ)—the successor to the lab at which Alan Turing worked during World War II. Unfortunately, his contributions, along with those of his colleagues Clifford Cocks and Malcolm Williamson, remained classified until 1997 and therefore had no influence on computer science during the years in which those ideas were locked away from public view. The benefits of public-key cryptography were unavailable until Martin Hellman, Whitfield Diffie, and Ralph Merkle reinvented the concept at Stanford University in 1976.

13.1 The one-time pad

Before describing the idea of public-key encryption in more detail, it makes sense to introduce a more traditional encryption scheme that is provably secure, even though it has other problems that limit its use in practice. The reason that codes can be broken in the first place is that ciphertext messages—encrypted though they are—contain information that codebreakers can exploit. A letter-substitution cipher, for example, preserves letter frequencies, which make such ciphers easy to decode. The messages transmitted by the Enigma machine described in Chapter 12 contained patterns determined by the structure of the rotors that made it possible to break the German codes. One way to create an unbreakable code is to use an encryption strategy that reveals no information at all, at least in a mathematical sense. If an encrypted message is indistinguishable from a random sequence of bits, the codebreaker has no information on which to build.

The simplest way to implement a code that conveys no information is to have the sender and receiver share a codebook that defines a transformation for each character in the message so that the transformation has the following properties:

• Each transformation is randomly chosen in a way that makes every possible encryption equally likely.
• No transformation is ever used more than once.

Because of the rule that no transformation is ever used more than once, this type of code is usually called a one-time pad, although it is sometimes referred to as the Vernam cipher after Gilbert Vernam, who invented the system in 1918 along with Joseph O’Mauborane.

The mechanics of the one-time pad are best illustrated by example. Suppose that you are a secret agent and need to send information back to your headquarters in a completely secure way. Before you leave, you take with you a codebook consisting of a sequence of random numbers, each of which is in the range 0 to 25. A duplicate copy of that codebook remains at headquarters, which is the only other
copy in existence. The list of numbers, moreover, must contain at least as many numbers as there are letters in all the messages you ever plan to send so that you will never have to reuse any of the numbers. Such codebooks must therefore have a large number of entries, but let’s imagine that your codebook begins like this:

```
3 24 8 15 4 25 17 7 14 2 21 18 21
```

Now suppose that you are on your mission and need to send the message **SEND HELP NOW** back to headquarters. To do so, you begin by copying down the letters in the message, ignoring any spaces or punctuation. Beside each letter in the plaintext, you then copy a number from the codebook, using each number exactly once. To form the ciphertext, you simply use the numbers from the codebook to advance the plaintext letters in the alphabet exactly as in the Caesar cipher. The first letter of the message is **S**, which becomes **V** when it is shifted three letters ahead in the alphabet. To get the next letter, you add 24 to **E**, which cycles back around through the beginning of the alphabet and becomes **C**. As shown in the diagram to the left, the encoding process generates the ciphertext **VCVSLDCWBQR**.

When headquarters receives the message, all it has to do is subtract the numbers in the codebook, restoring the original message.

It is not clear, however, why such a message would be secure. Anyone listening in on the communication channel can intercept the encrypted message. Why is it harder to decipher a message if it is sent using a one-time pad than if it is sent using another type of code?

The security of the one-time pad was established by Claude Shannon in 1949 as a natural consequence of his more general theory of information. The mathematical arguments he used to do so are beyond the scope of this book, but it is not hard to understand informally why messages coded using this scheme provide no useful information to the eavesdropper. Suppose that you intercept the encrypted message **VCVSLDCWBQR**. From what you have—even if you know the message was encoded using a one-time pad—what can you determine about the corresponding plaintext? Given the structure of the algorithm, you know that the plaintext is 11 letters long, but what are those 11 letters? Is there any way to find out?

The critical property that makes the one-time pad secure is that the letters in the ciphertext depend just as much on the random number entries in the codebook as they do on the plaintext message. The encrypted message **VCVSLDCWBQR** could have come from *any* set of plaintext letters at all, if the codebook had contained the appropriate sequence of random numbers. Suppose, for example, that you had been trying to send **MISSION DONE** instead, but had used a codebook starting with the random sequence 9, 20, 3, 0, 3, 15, 15, 19, 13, 3, 13. In that situation, you would produce exactly the same ciphertext, even though the plaintext is different.
For any given plaintext message, the letters in the resultant ciphertext are equally probable and—at least to an eavesdropper without access to the codebook—just as random as the numbers used to produce it. In an information-theoretic sense, the ciphertext produced by a one-time pad has the maximum possible entropy, which measures its degree of disorder. Such a message conveys no information and therefore represents a perfect code.

One-time pads have in fact been used to convey secret information. During his attempt to incite a revolution in Bolivia, Che Guevara used a one-time pad to exchange messages with Fidel Castro in Cuba; the secret codebook containing the random numbers was taken from Guevara when he was captured and executed in 1967. On the whole, however, the one-time pad has never enjoyed much popularity because of the following drawbacks:

1. The codebook needs to be incredibly large. The one-time pad derives its security from the fact that the numbers it contains are never reused, which ensures that every letter in the ciphertext is unrelated to any other. Unfortunately, this restriction means that the codebooks used in the process must be large enough to accommodate every message that is ever sent by the communicating parties.

2. The problem of distributing the codebook is as hard as sending messages. When using a one-time pad, the sender and receiver must share a common codebook that is itself secure. To do so, they must either meet face to face or depend on some reliable courier to exchange the codebooks. The former strategy is cumbersome, and the latter puts the codebook at risk of interception.

Because of these weaknesses, practical applications of cryptography usually rely on a much smaller amount of shared information between the sender and the receiver. Instead of a complete codebook in which each entry is used only once, most conventional encryption schemes rely on a relatively small secret key that is shared by the sender and the receiver. Suppose, for example, that you and your headquarters decide that you will make do with a much smaller codebook and reuse the same random numbers when you exhaust the original set. That reusable set of numbers would then become the key. The code is no longer mathematically unbreakable, and its security depends on the size of the key. If the key consists of a million numbers, the resulting code would certainly be secure for the first million characters, since it is only the reuse of the key values that offers any information. On the other hand, if the key were only a single number, the resulting code would be a simple Caesar cipher, which any eavesdropper could easily break.

In general, longer encryption keys provide greater security. As computing machines get faster, however, the size of a key needed to ensure reasonable security increases as well. When IBM designed the Data Encryption Standard (DES) in the
1970s, its 56-bit key seemed secure. With the computers that are available today, that level of security is no longer adequate. New encryption standards that rely on secret keys now tend to use 128- or even 256-bit keys. These are secure for the moment, but it is not clear how long they will be safe given advances in technology.

Even so, having to find secure ways to transmit keys is certainly an annoyance.

### 13.2 The idea of public-key encryption

The primary advantage of public-key cryptography is that it eliminates the problem of distributing secret keys. Since the keys are public—as the name of the technology implies—it is perfectly reasonable to send them over an insecure channel or even publish them in a directory. Initially, such an idea seems absurd. If everyone can see the key, then anyone could presumably decipher messages sent using it. As reasonable as it sounds, however, this intuition is not in fact true. Recognizing that secure communication can still occur even after revealing at least a part of the key is the insight that makes public-key encryption so powerful.

The idea that a coded message can be secure even though the keys are public is certainly counterintuitive. In a formerly classified paper from 1970, Ellis writes

> It is generally regarded as self-evident, that, in order to prevent an interceptor from understanding a message which is intelligible to the authorised recipient, it is necessary to have some initial information known to the sender and to the recipient but kept secret from the interceptor.

To understand why this seemingly obvious principle might in fact be wrong, it helps to take a step back and think more abstractly about the problem of sending an encrypted message and what it means for that communication to be secure.

**Introducing Alice and Bob**

At its essence, secure communication is the process of sending a message from one individual to another in a way that ensures that the contents of that message cannot be deciphered by an eavesdropper somewhere along the way. There are therefore three entities to consider—a sender, a receiver, and an eavesdropper—each of whom appears in Figure 13-1. Following the usual convention in cryptographic literature, I’ll call the communicating parties Alice and Bob. Since John Tenniel has provided us with a delightful Alice in his illustrations for Lewis Carroll’s *Alice and Wonderland*, I’ll assume that the Mad Hatter’s given name is Bob and use him as the receiver. As Alice transmits a message to Bob, it is susceptible to interception by an eavesdropper, often introduced as a character named Eve, but represented here by the Cheshire Cat, whose capacity to disappear makes him devious enough to play the role.
Limitations on the eavesdropper

One important question to resolve when using this model is what restrictions, if any, exist on the power of the eavesdropper. To err on the side of security, it is best to assume that the eavesdropper has access to the best technology and is resourceful enough to intercept messages that pass through any insecure space. At the same time, there must be some things that the eavesdropper cannot do. If an eavesdropper, for example, could read Alice’s mind or watch through solid walls as Bob transcribed the received message, it wouldn’t matter what encryption system Alice and Bob chose to use. Without some limitations on the eavesdropper, the cryptography problem becomes meaningless. Traditional analysis of cryptographic communication therefore assumes that the sender and the receiver have a security zone within which they can maintain secrecy. Alice, for example, can keep information secret as long as she never transmits it or otherwise reveals it to anyone. It is only when information passes outside the security zone—indicated by the dashed lines in Figure 13-1—that it becomes available to other parties.

If Alice sends a message to Bob in its plaintext form, the eavesdropper—who can intercept anything that passes outside the security zone—would not need to decipher it at all. If Alice uses a Caesar cipher or a simple letter-substitution code, any competent eavesdropper would be able to decode the message quickly, even without computer assistance. If Alice uses a polyalphabetic code in the manner of the Enigma machine described in Chapter 12, breaking the code would take longer. Given the enormous power of computers today, however, Alice could have no assurance that no one would be able to intercept and decipher her message.
The structure of public-key encryption

The idea behind public-key encryption is actually quite simple to express. In their landmark paper published in the *IEEE Transactions on Information Theory* in November 1976, Whitfield Diffie and Martin Hellman describe the basic principle as follows:

In a public key cryptosystem enciphering and deciphering are governed by distinct keys, $E$ and $D$, such that computing $D$ from $E$ is computationally infeasible. . . . The enciphering key $E$ can thus be publicly disclosed without compromising the deciphering key $D$. Each user of the network can, therefore, place his enciphering key in a public directory. This enables any user of the system to send a message to any other user enciphered in such a way that only the intended receiver is able to decipher it. . . . A private conversation can therefore be held between any two individuals regardless of whether they have ever communicated before. Each one sends messages to the other enciphered in the receiver’s public enciphering key and deciphers the message he receives using his own secret deciphering key.

Figure 13-2 on the next page illustrates the steps involved in this process. Before anyone can send him a message, Bob must create a pair of keys. He makes public his encryption key, which is labeled $E_b$ in Figure 13-2, but keeps the decryption key $D_b$ secret. The public key can be seen and used by anyone, including both Alice and the Cheshire Cat.

The second step occurs when Alice decides that she wants to send a message to Bob. She looks up Bob’s public encryption key and uses $E_b$ to convert her message into a coded sequence of bits. She then sends the coded message to Bob using any communication channel she chooses. The Cheshire Cat might well be listening in, but would nonetheless be unable to decipher the message. Deciphering the message requires the application of $D_b$, which has never left Bob’s security zone. As a result, only Bob can decipher the message, as shown in the third step of the process.

The role of one-way functions

The piece of the process that you are most likely to find unsettling at this point is the idea that knowing how to encrypt a message—which everyone knows in a public-key system—provides no help in decrypting that message. For the cryptographic techniques described in Chapter 12, having the encryption key is a dead giveaway to the decryption problem. If Turing had actually had the codebook that served as the key for the Enigma machine, all of the wartime work done at Bletchley Park would have been unnecessary. For traditional private-key systems, having the key makes it easy to “invert” the encryption process, allowing an eavesdropper to translate coded messages back to their plaintext originals.
Figure 12.2 Steps in a public-key exchange

Step 1. Bob creates two keys: a private key $D_B$ and a public key $E_B$ visible to everyone, including the Cheshire Cat.

Step 2. Alice uses the public key $E_B$ to encrypt a message to Bob. The Cheshire Cat can’t read it without $D_B$.

Step 3. Bob decodes the message using the private decryption key $D_B$. 

Twas brillig, and...

10110000100111101010001010111
In general, the problem of inverting an encoding transformation may not be all that easy. In a public-key cryptosystem, you can’t use just any process to come up with the keys $D$ and $E$. As Diffie and Hellman note in their paper, it must also be the case that “computing $D$ from $E$ is computationally infeasible.” Mathematically based encryption functions that are easy to apply in one direction but nearly impossible to invert are called one-way functions. These one-way functions are sometimes called trapdoor functions because they make it easy to drop down but hard to get back up.

Suggesting that one-way functions might be useful in cryptography, however, is an easier task than finding one that works well in practice. Diffie and Hellman initially proposed a scheme in which the difficulty of inverting the function used for encryption derived from its similarity to a computationally difficult problem called the subset-sum problem, which is one of the $\textbf{NP}$-complete problems described in Chapter 11. The implementation they used, however, differs in several respects from the classic subset-sum problem and turns out to be much easier to solve. Their paper, however, encouraged other computer scientists to develop public-key techniques. A team of computer scientists at MIT took up the challenge and found a more effective strategy. The MIT approach, which has since become the dominant implementation of the public-key concept, is described in the following section.

13.3 The RSA algorithm

In 1977, three computer scientists then at MIT—Ron Rivest, Adi Shamir, and Len Adleman— invented a new implementation of public-key cryptography, which has come to be known as the RSA algorithm after the initial letters of its inventors’ names. The first published description of RSA appeared in Martin Gardner’s “Mathematical Games” column in the August 1977 issue of Scientific American, which was soon followed by a more technical presentation in the February 1978 issue of Communications of the ACM.

Generating keys

As in any public-key encryption scheme, RSA requires that anyone wishing to receive a message have a pair of keys: a public key used by others to encrypt messages and a private key that allows the recipient to decrypt that message. In RSA, generating the keys consists of the following steps:

1. Select two prime numbers, $p$ and $q$.
2. Set $n$ equal to $pq$ and $t$ equal to $(p-1)(q-1)$.
3. Choose an integer $d$ that is less than $n$ and has no common factors with $t$.
4. Compute a new integer $e$ with the property that the remainder of $d \times e$ divided by $t$ is 1. This number is called the modular inverse of $d$ with respect to $t$. 

Ron Rivest

Adi Shamir

Len Adleman
Once you have performed these steps, you release \( e \) and \( n \) as your public key and keep \( d \) hidden as your private key. To avoid making it easy for others to discover the value of \( d \), you must not reveal the values \( p \), \( q \), and \( t \) used in the key generation process, although you won’t need these values to encode and decode messages.

To maximize the security of your keys, \( p \) and \( q \) should be large integers with 200 or more digits each. Numbers of that size, however, make examples harder to follow, particularly when the numbers don’t fit on a single line. To understand the process, it is better to pick smaller values for \( p \) and \( q \) that are nonetheless large enough to construct a meaningful example. The example that follows uses the following seven-digit values, both of which are prime:

\[
\begin{align*}
p &= 1089653 \\
q &= 7834529
\end{align*}
\]

Given these values for \( p \) and \( q \), the two new values generated in step 2 are \( n \), which is \( p \times q \), and \( t \), which is \((p - 1) \times (q - 1)\). Given the sample values for \( p \) and \( q \), these derived parameters have the following values:

\[
\begin{align*}
n &= 8536918028437 \\
t &= 8536909104256
\end{align*}
\]

Step 3 calls for choosing an integer \( d \) less than \( n \) having no common factors with \( t \). One way to guarantee that \( d \) and \( t \) have no common factors is to choose a prime value for \( d \), such as

\[
d = 2567481675497
\]

The final step in the key generation process consists of finding the modular inverse of \( d \) with respect to \( t \). Finding modular inverses turns out to be relatively simple computationally. An extension of Euclid’s algorithm for calculating the greatest common divisor from the 5th century BCE does the job nicely. The details of that extension are not important to an understanding of the RSA algorithm but are easy to find on the web.

If you apply this function to the value of \( d \), you find that the modular inverse is 3348048448985.

Thus, at the end of the process, the public parts of the key are

\[
\begin{align*}
n &= 8536918028437 \\
e &= 3348048448985
\end{align*}
\]

and the private key is

\[
d = 2567481675497
\]
Sending and receiving messages

Suppose that you want to send someone an encrypted message using the RSA algorithm. The first step is to convert your message into an integer value. Given that computers already use numeric codes to represent characters, this process is extremely easy. All you need to do is take the sequence of bits represented by the characters in the message and see what value those bits correspond to as a number. For example, suppose that you wanted to transmit the string “Hello” as a message. Assuming that the characters are represented in ASCII, that string consists of the following 40 bits:

\[ \begin{array}{cccccccc}
\text{H} & \text{e} & \text{l} & \text{l} & \text{o} \\
01001000 & 01101011 & 01101100 & 01101110 & 01101111
\end{array} \]

If you convert this sequence of bits to an integer, you get the numerical value of the plaintext message, as follows:

\[ m = 310939249775 \]

The values that you encode using the RSA algorithm cannot be larger than the value of \( n \) that is distributed as part of the public key. If they are, you need to break the message into blocks so that the numeric equivalent of each block remains less than \( n \). Because \( n \) is typically more than 200 digits long, you can encode many characters in a single block. The value of \( n \) from the preceding section is nowhere near that large, but it is larger than the numerical value of the five-character string “Hello”, which means you can send this message as a single unit.

Once you have created the keys, the first step in the process of creating an encrypted RSA message is to use the numerical value of the message \( m \) to create the encrypted message \( c \) using the following formula:

\[ c = m^e \mod n \]

In this formula, the word “\( \mod \)” indicates the remainder operation. In most modern programming languages, including JavaScript, the remainder operation is indicated using the % operator. If you read papers on cryptography, however, you will typically see this operator written out as a word, which matches its conventional mathematical form.

Although performing the necessary calculation in an efficient way requires a bit of mathematical cleverness, the encryption formula gives the following value for \( c \):

\[ c = m^e \mod n = 6573003815055 \]
To decrypt the message, all that the receiver needs to do is use the private decryption key $d$ in a similar calculation, which gives the following result:

$$m' = c^d \mod n = 310939249775$$

Happily, the value of the decrypted message $m'$ matches the original message $m$, which means that the original message has gotten through to the receiver.

13.4 The mathematics of RSA

As with any public-key system, the RSA algorithm described in the preceding section depends on the fact that the decryption operation inverts the encryption operation, thereby restoring the original message. If the sender computes the ciphertext $c$ by evaluating

$$c = m^e \mod n$$

the receiver must be able to restore the original message by computing

$$m = c^d \mod n$$

If all you want to do is use the RSA encryption system, you can simply trust the claim that the process used to select $d$, $e$, and $n$ ensures that this relationship holds. Understanding why it does so, however, is interesting in its own right, partly because it draws on a range of mathematical work stretching back to antiquity. The sections that follow describe the mathematics behind the RSA algorithm and offer an important illustration of how theory contributes to practice.

Prime numbers

The first concept that you need to understand is that of a prime number, which is a central concept in a branch of mathematics known as number theory. The sections that follow make use of the following definitions, some of which are a review of the earlier presentation:

- A **prime number** is an integer greater than 1 that is divisible only by itself and 1. For example, the integers 2, 3, 5, 7, and 11 are prime because they have no factors other than themselves and 1. By contrast, the integer 6 can be expressed as $2 \times 3$, and is therefore **composite**.

- Two integers are **relatively prime** if they have no common factors other than 1. For example, the integers 40 and 63 are relatively prime because there are no integers other than 1 that divide evenly into both. The integers 39 and 63 are not relatively prime because they share 3 as a common factor.
Modular arithmetic

Understanding the mathematics of the RSA algorithm depends on having some familiarity with the idea of modular arithmetic, which is a simplified form of integer arithmetic in which all results are reduced to a finite range by dividing them by a fixed integer called the modulus. For example, if you are using modular arithmetic with a modulus of 12, the sum of 6 and 8 is not 14 but 2, which is the remainder left over when 14 is divided by 12. In mathematics, this relationship is denoted as

\[ 6 + 8 = 2 \pmod{12} \]

The triple equal sign in this formula stands for the words *is congruent to*. The entire expression can therefore be read as *six plus eight is congruent to two, mod twelve*. The word *congruent* is used to signify that the two sides of the = relationship may not be equal but instead have the same remainder when divided by the modulus.

Outside of formal mathematics, modular arithmetic is sometimes called *clock arithmetic* because its operations resemble calculation on a clock. If, for example, you add eight hours to six o’clock, the resulting time is two o’clock because the numbers on the clock cycle back to the beginning after going past twelve. Clocks therefore implement modular arithmetic with a modulus of 12. Modular arithmetic also comes up in calculations involving days of the week, which require computation with a modulus of 7.

Although the idea of modular arithmetic is extremely old and shows up in early Egyptian, Greek, Chinese, Indian, and Mayan cultures, the systematic treatment of the topic is usually credited to the German mathematician Carl Friedrich Gauss, who included an extensive discussion of modular arithmetic in his *Disquisitiones Arithmeticae*, published in 1801.

Understanding the RSA algorithm requires you to learn only a few properties of modular arithmetic, of which the following two are the most important:

1. If \( a \equiv b \pmod{n} \), then there must be some integer \( k \) for which \( a = b + k \times n \). This property simply restates the idea of congruence and follows directly from the definition of a remainder.
2. If \( a \equiv b \pmod{n} \) and \( c \equiv d \pmod{n} \), each of the following congruence relationships is true:

\[ a + c \equiv b + d \pmod{n} \]
\[ a - c \equiv b - d \pmod{n} \]
\[ a \times c \equiv b \times d \pmod{n} \]
The practical effect of the second set of rules is that modular arithmetic allows you to retain only the remainder when you apply the standard addition, subtraction, and multiplication operations. Even if you are in the middle of a long series of operations, you can simplify the calculation by dividing your intermediate results by the modulus and throwing away everything except the remainder. The final result you get using this approach will always match the answer you would have gotten if you carried out the full calculation and then took the remainder at the end.

To get a sense of how important this simplification can be, try to imagine how you would go about calculating the value of \( m^e \) where \( m \) is 310939249775 and \( e \) is 3348048448985. If you try to compute the result simply by repeated multiplication, you quickly discover that the numbers quickly become difficult to manage. The first three steps in this series of multiplications look like this:

\[
\begin{align*}
m &= 310939249775 \\
n^2 &= 96683217050639837550625 \\
n^3 &= 3006260697555943921719211082359375 \\
n^4 &= 9347644459261133808068043868403689799437890625 
\end{align*}
\]

After just four multiplication steps, the result already has 46 digits. If you were somehow able to continue this process through all 3348048448985 multiplications, the final product would have more than 38 trillion digits, which means that you would need a hefty computer just to store that value. The calculation, moreover, would take an enormous amount of time even on the fastest processors. If RSA required this much space and time, no one would think of using it.

Fortunately, there are two clever mathematical strategies that you can use to make this computation much more manageable. The first is modular arithmetic, which allows you to reduce the size of the intermediate results. If you look back at the RSA encryption formula

\[
c = m^e \mod n
\]

you’ll see that you don’t need the actual value of \( m^e \); all you need is the remainder when you divide by \( n \). Modular arithmetic allows you to divide by \( n \) on each cycle, which means that you never have to work with any values larger than \( n \). Keeping only the remainder makes the results of the repeated multiplication much shorter:

\[
\begin{align*}
m &= 310939249775 \pmod{n} \\
n^2 &= 5749655927223 \pmod{n} \\
n^3 &= 1507464499484 \pmod{n} \\
n^4 &= 7261499321870 \pmod{n}
\end{align*}
\]

Even using modular arithmetic, carrying out these repeated multiplication all the way to \( m^{3348048448985} \) will take more time than anyone is willing to spend. To achieve
the next level of efficiency, you need to use a faster algorithm for raising a number to a power. As it turns out, it is straightforward to implement a raise-to-power function that runs in \(O(\log N)\) time, which offers just the kind of speedup needed for this application. Instead of taking some three trillion multiplications, the improved implementation requires on the order of \(\log_2 348048448985\) multiplications, which is a much more manageable 42.

Even though the numbers in this example are smaller than they would be in practice, they are too large to carry out the computations by hand. Try starting instead with the following smaller values:

\[
\begin{align*}
    p &= 13 \\
    q &= 23 \\
    d &= 17
\end{align*}
\]

Apply the techniques from this chapter to compute the value of encryption key \(e\), along with the derived parameters \(n\) and \(t\). Use these values to encrypt the single-letter message "A" (ASCII 65). Show that applying the decryption key \(d\) to the resulting ciphertext gives you back the letter "A".

**The Euler totient function**

In the early 1700s, the Swiss mathematician Leonhard Euler (pronounced oiler) discovered several important properties that link the ideas of prime numbers and modular arithmetic. In particular, Euler identified an important mathematical function that plays a central role in the mathematics of RSA. That function is called the *Euler totient function* and is denoted by the Greek letter \(\phi\) (which mathematicians usually pronounce as fee, even though fie is more conventional in other disciplines). The Euler totient function is defined as follows:

\[
\phi(n) = \text{the number of integers between 1 and } n \text{ that are relatively prime to } n
\]

For small values of \(n\), the Euler totient function is easy to compute. Suppose, for example, that \(n = 12\). If you write out the integers between 1 and 12 and then cross out the numbers that share a common factor with 12, you will see that only four numbers are left, as follows:

\[
\begin{array}{cccccccccc}
    \text{1} & \times & \times & \times & \text{5} & \times & \text{7} & \times & \times & \times \\
\end{array}
\]

Thus, \(\phi(12) = 4\). If \(n = 13\), you would be able to cross out only the number 13 itself, which means that \(\phi(13) = 12\), as shown:

\[
\begin{array}{ccccccccccc}
    \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} & \text{9} & \text{10} & \text{11} & \text{12} & \times \\
\end{array}
\]
The table at the right shows the value of the Euler totient function for the first 15 integers, all of which are easy to compute by hand. As the value of \( n \) gets larger, however, computing the value of \( \phi(n) \) begins to take much longer. For numbers with a hundred or more digits, it is computationally prohibitive to compute \( \phi(n) \) by checking every possible value between 1 and \( n \) to see whether it should be included in the count. There are just too many values to check.

There are, however, two important special cases in which computing the totient function is quite easy, even for large numbers. The first occurs when the number itself is prime. For any prime number \( p \), every integer less than \( p \) is relatively prime to \( p \), which means that

\[
\phi(p) = p - 1
\]

This relationship is illustrated by the table of values in the margin, which shows, for example, that \( \phi(7) = 6 \), \( \phi(11) = 10 \), and \( \phi(13) = 12 \).

The second case—which is the critical one for the RSA algorithm—occurs when \( n \) is the product of two distinct primes \( p \) and \( q \). In this case, every integer less than \( n \) is relatively prime to \( n \), except those that are divisible by \( p \) and those divisible by \( q \). If you think for a moment about the numbers between 1 and \( pq \) that are divisible by \( q \), you will realize that there are \( p \) such values, as follows:

\[
q, 2q, 3q, \ldots, (p-2)q, (p-1)q, pq
\]

Symmetrically, in the positive integers less than \( pq \), there are \( q \) multiples of \( p \). Both lists, however, include the number \( pq \), which means that it gets counted twice. The number of integers between 1 and \( pq \) that are relatively prime to \( pq \) can therefore be expressed as follows:

\[
\phi(pq) = pq - p - q + 1 = (p-1)(q-1)
\]

Thus, for any two distinct primes \( p \) and \( q \), \( \phi(pq) \) is simply \( (p-1)(q-1) \). As an example, \( \phi(15) = \phi(3 \times 5) = (3-1)(5-1) = 2 \times 4 = 8 \).

If you look back at the section entitled “Generating keys” earlier in this chapter, you will see that the value \( r \) is \( (p-1)(q-1) \). Thus, \( r \) is equal to \( \phi(n) \). This fact will turn out to be the critical factor in unraveling the RSA puzzle.

**Euler’s theorem**

What makes Euler’s totient function interesting is the relationship between \( \phi(n) \) and modular congruence. Sometime around 1750, Euler proved the following theorem:

**Euler’s theorem:** If \( m \) and \( n \) are relatively prime, then \( m^{\phi(n)} \equiv 1 \pmod{n} \).
The proof of this theorem is beyond the scope of this book. Even so, it is easy to pick values for $m$ and $n$ and check that the theorem works for those values. The table on the left, for example, checks the theorem for all possible values of $m$ when $n$ is 15, which means that $\phi(n)$ is 8. The list of congruences shows that $m^8 \equiv 1 \pmod{15}$ only for those values—1, 2, 4, 7, 8, 11, 13, and 14—that are relatively prime to 15. For those values—3, 5, 6, 9, 10, and 12—that share a common factor with 15, $m^8 \pmod{15}$ leaves some other remainder.

By itself, Euler’s theorem is not sufficient to establish the correctness of the RSA algorithm. To be sure that the RSA system can transmit messages whose numeric representation happens to be divisible by one of the primes chosen as $p$ and $q$ in the key generation process, it is necessary to use the following somewhat stronger theorem, which generalizes Euler’s result:

**Generalized version of Euler’s theorem:** If $n$ is the product of distinct primes, $p$ and $q$, then $m^{\phi(n)+1} \equiv m \pmod{n}$ for any nonnegative integer $m < n$ and for any integer $k$.

Once again, illustrations of this more general theorem are easy to provide, even though the proof remains beyond the scope of the text. The table to the left shows, for example, that raising any integer less than 15 to the 9th power always leaves the original integer as a remainder when the result is divided by 15. The reason that the ninth power has this property is simply that 9 is one more than 8, which is $\phi(15)$. The same result holds for the 17th power, the 25th power, or any other power that is one more than an integral multiple of $\phi(15)$.

**The correctness of RSA**

For public-key systems to encrypt and decrypt messages, it must be the case that applying the encryption key to a message and then applying the decryption key to the result restores the original message. This relationship can be expressed in functional form as follows:

$$ m = D(E(m)) $$

In RSA, the encryption function $E$ consists of the operation

$$ E(m) = m^e \pmod{n} $$

for the public-key values $e$ and $n$. The decryption function $D$ is essentially the same, but uses the secret value $d$ in place of $e$:

$$ D(m) = m^d \pmod{n} $$

If you put the two operations together, you get the following equation, given that you can move the remainder operation to the end of the calculation:
By the laws of exponents, this last equation is equivalent to

\[ D(E(m)) = m^{ed} \mod n \]

It is at this point that the process used to select \( e \) and \( d \) becomes important. The values \( d \) and \( e \) were chosen to be modular inverses with respect to \( \phi(n) \), which means that

\[ ed = 1 \pmod {\phi(n)} \]

This relationship in turn implies that

\[ ed = k\phi(n) + 1 \]

for some integer \( k \). Thus, the value of the encrypted-and-then-decrypted message can be expressed as

\[ D(E(m)) = m^{k\phi(n)+1} \mod n \]

From here, the generalized version of Euler’s theorem from the preceding section provides the one remaining step. As long as \( n \) is the product of distinct primes and \( m \) is less than \( n \)

\[ m^{\phi(n)+1} \equiv m \pmod n \]

The left side of this congruence matches precisely the value of the message after the encryption and decryption operations have been performed, which means that

\[ D(E(m)) = m \]

This equation shows that the RSA decryption operation does indeed invert the encryption operation.

**The security of RSA**

The preceding section shows that the RSA algorithm can be used to encrypt and decrypt a message. It says nothing, however, about whether that process is secure. If an eavesdropper could somehow deduce the secret value \( d \) from the publicly available values \( e \) and \( n \), the security would vanish. The value \( d \) would be easy to compute if the eavesdropper had access to any of the quantities—\( p, q, \) or \( \phi(n) \)—that were used to compute it. The important question is whether it would be possible to figure out \( d \) without these values.

Given the known information, the value of \( d \) is certainly computable. If you could factor \( n \) into its two component primes or if you had the patience to count all the integers between 1 and \( n \) that are relatively prime to \( n \), you’d have the answer.
When RSA is used in practice, however, the value \( n \) typically has 200 or more digits. Factoring a number of that size is a very difficult problem that—at least for the moment—satisfies the criterion proposed by Diffie and Hellman that, for a pair of encryption and decryption keys \( D \) and \( E \), “computing \( D \) from \( E \) is computationally infeasible.” The critical question for the RSA cryptosystem, however, is whether developments in mathematics will eventually enable codebreakers to factor the huge numbers on which the coding system depends.

In recent years, progress in factoring techniques has been extremely rapid. From 1991 to 2007, RSA Data Securities—the company that commercialized the RSA techniques—sponsored a factoring challenge in which they offered cash prizes to anyone who could factor numbers of varying lengths. The initial values in the challenge were numbered according to the number of digits, so that RSA-100, for example, has 100 decimal digits. Later additions to the list were numbered according to the number of bits, so that RSA-576 is 567 bits long.

Very soon after the challenge was announced, Mark Manasse and Arjen Lenstra factored RSA-100, which was shown to be the product of the primes

\[
40094690950920881030683735292761468389214899724061
\]

and

\[
37975227936943673922808872755445627854565536638199
\]

Since that time, several additional numbers from the RSA list have fallen to the efforts of mathematicians and computer scientists throughout the world, as shown in the table on the left. As mathematicians find ways to factor ever larger integers, the number of bits needed for secure encryption grows accordingly. For the moment, keys with the currently recommended size of 1024 bits remain safe, although they may not be in the future.

<table>
<thead>
<tr>
<th>RSA factoring milestones</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA-100 1991</td>
</tr>
<tr>
<td>RSA-110 1992</td>
</tr>
<tr>
<td>RSA-120 1993</td>
</tr>
<tr>
<td>RSA-130 1996</td>
</tr>
<tr>
<td>RSA-140 1999</td>
</tr>
<tr>
<td>RSA-155 1999</td>
</tr>
<tr>
<td>RSA-160 2003</td>
</tr>
<tr>
<td>RSA-170 2009</td>
</tr>
<tr>
<td>RSA-180 2010</td>
</tr>
<tr>
<td>RSA-190 2010</td>
</tr>
<tr>
<td>RSA-200 2005</td>
</tr>
<tr>
<td>RSA-576 2003</td>
</tr>
<tr>
<td>RSA-640 2005</td>
</tr>
<tr>
<td>RSA-704 2012</td>
</tr>
<tr>
<td>RSA-768 2009</td>
</tr>
</tbody>
</table>

### 13.5 Digital signatures

The problem of an eavesdropper listening in on a conversation is not the only threat to secure communications. If the Cheshire Cat wanted to sabotage the communication between Alice and Bob, it could do so by forging messages to Bob, making it seem as if those messages came from Alice. Bob’s key, after all, is public. That means that anyone can send Bob an encrypted message. If Bob receives a message that claims to be from Alice, how can he be sure that Alice is the source and not the Cheshire Cat pretending to be Alice?

As it turns out, public-key encryption can solve this problem as well. The basic idea is for Alice to sign her messages in a way that nobody but Alice could, because to do so they would have to know Alice’s secret key. Strategies that allow senders to establish their identity are called digital-signature schemes.
Most digital signature schemes depend on the fact that public-key encryption algorithms typically require both the encryption and decryption functions to act as inverses for the other. To be an encryption scheme at all, it must be the case that

\[ D(E(m)) = m \]

since it is this property that allows the receiver to decode the message. Most encryption schemes, however, are designed so that it also the case that

\[ E(D(m)) = m \]

This property holds for RSA, because it doesn’t matter in which order you perform the exponentiation. Thus,

\[ E(D(m)) = (m^d)^e \mod n = (m^e)^d \mod n = D(E(m)) \]

To sign a message, all Alice needs to do is apply her secret decryption key \( D_A \) to her name and then include that coded signature at the end of her message to Bob. As with any encoded text, the signature is simply a string of bits. Because Bob knows Alice’s public encryption key \( E_A \), however, he can turn those bits back into a name by applying \( E_A \) to the coded signature. If the name comes out as “Alice,” Bob knows that only someone with Alice’s secret key could possibly have sent the message. The entire process is illustrated in Figure 13-3.
It’s coming quickly, and if I live to be old I shall see the sky as pestilential as the roads. It really is a new civilisation. I have been born at the end of the age of peace and can’t expect to feel anything but despair. Science instead of freeing man—the Greeks nearly freed him by right feeling—is enslaving him to machines. Nationality will go, but the brotherhood of man will not come . . .

God, what a prospect! The little houses that I am used to will be swept away, the fields will stink of petrol, and the airships will shatter the stars. Man may get a new and perhaps greater soul for the new condition. But such a soul as mine will be crushed out.

—letter written by E. M. Forster, 1908

The Machine Stops

—E. M. Forster, 1909

1 THE AIR-SHIP

Imagine, if you can, a small room, hexagonal in shape, like the cell of a bee. It is lighted neither by window nor by lamp, yet it is filled with a soft radiance. There are no apertures for ventilation, yet the air is fresh. There are no musical instruments, and yet, at the moment that my meditation opens, this room is throbbing with melodious sounds. An armchair is in the centre, by its side a reading-desk—that is all the furniture. And in the armchair there sits a swaddled lump of flesh—a woman, about five feet high, with a face as white as a fungus. It is to her that the little room belongs.

An electric bell rang.

The woman touched a switch and the music was silent.

“I suppose I must see who it is,” she thought, and set her chair in motion. The chair, like the music, was worked by machinery and it rolled her to the other side of the room where the bell still rang importunately.

“Who is it?” she called. Her voice was irritable, for she had been interrupted often since the music began. She knew several thousand people, in certain directions human intercourse had advanced enormously.

But when she listened into the receiver, her white face wrinkled into smiles, and she said:

“Very well. Let us talk, I will isolate myself. I do not expect anything important will happen for the next five minutes—for I can give you fully five minutes, Kuno. Then I must deliver my lecture on ‘Music during the Australian Period’.”

She touched the isolation knob, so that no one else could speak to her. Then she touched the lighting apparatus, and the little room was plunged into darkness.

“Be quick!” She called, her irritation returning. “Be quick, Kuno; here I am in the dark wasting my time.”

But it was fully fifteen seconds before the round plate that she held in her hands began to glow. A faint
blue light shot across it, darkening to purple, and presently she could see the image of her son, who lived on the other side of the earth, and he could see her.

“Kuno, how slow you are.”

He smiled gravely.

“I really believe you enjoy dawdling.”

“I have called you before, mother, but you were always busy or isolated. I have something particular to say.”

“What is it, dearest boy? Be quick. Why could you not send it by pneumatic post?”

“Because I prefer saying such a thing. I want—”

“Well?”

“I want you to come and see me.”

Vashti watched his face in the blue plate.

“But I can see you!” she exclaimed. “What more do you want?”

“I want to see you not through the Machine,” said Kuno. “I want to speak to you not through the wearisome Machine.”

“Oh, hush!” said his mother, vaguely shocked. “You mustn’t say anything against the Machine.”

“Why not?”

“One mustn’t.”

“You talk as if a god had made the Machine,” cried the other.

“I believe that you pray to it when you are unhappy. Men made it, do not forget that. Great men, but men. The Machine is much, but it is not everything. I see something like you in this plate, but I do not see you. I hear something like you through this telephone, but I do not hear you. That is why I want you to come. Pay me a visit, so that we can meet face to face, and talk about the hopes that are in my mind.”

She replied that she could scarcely spare the time for a visit.

“The air-ship barely takes two days to fly between me and you.”

“I dislike air-ships.”

“Why?”

“I dislike seeing the horrible brown earth, and the sea, and the stars when it is dark. I get no ideas in an air-ship.”

“I do not get them anywhere else.”

“What kind of ideas can the air give you?”

He paused for an instant.
“Do you not know four big stars that form an oblong, and three stars close together in the middle of the oblong, and hanging from these stars, three other stars?”

“No, I do not. I dislike the stars. But did they give you an idea? How interesting; tell me.”

“I had an idea that they were like a man.”

“I do not understand.”

“The four big stars are the man’s shoulders and his knees. The three stars in the middle are like the belts that men wore once, and the three stars hanging are like a sword.”

“A sword?”

“Men carried swords about with them, to kill animals and other men.”

“It does not strike me as a very good idea, but it is certainly original. When did it come to you first?”

“In the air-ship—” He broke off, and she fancied that he looked sad. She could not be sure, for the Machine did not transmit nuances of expression. It only gave a general idea of people—an idea that was good enough for all practical purposes, Vashti thought. The imponderable bloom, declared by a discredited philosophy to be the actual essence of intercourse, was rightly ignored by the Machine, just as the imponderable bloom of the grape was ignored by the manufacturers of artificial fruit. Something ‘good enough’ had long since been accepted by our race.

“The truth is,” he continued, “that I want to see these stars again. They are curious stars. I want to see them not from the air-ship, but from the surface of the earth, as our ancestors did, thousands of years ago. I want to visit the surface of the earth.”

She was shocked again.

“Mother, you must come, if only to explain to me what is the harm of visiting the surface of the earth.”

“No harm,” she replied, controlling herself. “But no advantage. The surface of the earth is only dust and mud, no advantage. The surface of the earth is only dust and mud, no life remains on it, and you would need a respirator, or the cold of the outer air would kill you. One dies immediately in the outer air.”

“I know; of course I shall take all precautions.”

“And besides—”

“Well?”

She considered, and chose her words with care. Her son had a queer temper, and she wished to dissuade him from the expedition.

“It is contrary to the spirit of the age,” she asserted.

“Do you mean by that, contrary to the Machine?”

“In a sense, but—”

His image in the blue plate faded.

“Kuno!”
He had isolated himself.

For a moment Vashti felt lonely.

Then she generated the light, and the sight of her room, flooded with radiance and studded with electric buttons, revived her. There were buttons and switches everywhere—buttons to call for food for music, for clothing. There was the hot-bath button, by pressure of which a basin of (imitation) marble rose out of the floor, filled to the brim with a warm deodorized liquid. There was the cold-bath button. There was the button that produced literature. and there were of course the buttons by which she communicated with her friends. The room, though it contained nothing, was in touch with all that she cared for in the world.

Vashanti’s next move was to turn off the isolation switch, and all the accumulations of the last three minutes burst upon her. The room was filled with the noise of bells, and speaking-tubes. What was the new food like? Could she recommend it? Has she had any ideas lately? Might one tell her one’s own ideas? Would she make an engagement to visit the public nurseries at an early date?—say, this day this month.

To most of these questions she replied with irritation—a growing quality in that accelerated age. She said that the new food was horrible. That she could not visit the public nurseries through press of engagements. That she had no ideas of her own but had just been told one—that four stars and three in the middle were like a man: she doubted there was much in it. Then she switched off her correspondents, for it was time to deliver her lecture on Australian music.

The clumsy system of public gatherings had been long since abandoned; neither Vashti nor her audience stirred from their rooms. Seated in her armchair she spoke, while they in their armchairs heard her, fairly well, and saw her, fairly well. She opened with a humorous account of music in the pre Mongolian epoch, and went on to describe the great outburst of song that followed the Chinese conquest. Remote and primeval as were the methods of I-San-So and the Brisbane school, she yet felt (she said) that study of them might repay the musicians of today: they had freshness; they had, above all, ideas. Her lecture, which lasted ten minutes, was well received, and at its conclusion she and many of her audience listened to a lecture on the sea; there were ideas to be got from the sea; the speaker had donned a respirator and visited it lately. Then she fed, talked to many friends, had a bath, talked again, and summoned her bed.

The bed was not to her liking. It was too large, and she had a feeling for a small bed. Complaint was useless, for beds were of the same dimension all over the world, and to have had an alternative size would have involved vast alterations in the Machine. Vashti isolated herself—it was necessary, for neither day nor night existed under the ground—and reviewed all that had happened since she had summoned the bed last. Ideas? Scarcely any. Events—was Kuno’s invitation an event?

By her side, on the little reading-desk, was a survival from the ages of litter—one book. This was the Book of the Machine. In it were instructions against every possible contingency. If she was hot or cold or dyspeptic or at a loss for a word, she went to the book, and it told her which button to press. The Central Committee published it. In accordance with a growing habit, it was richly bound.

Sitting up in the bed, she took it reverently in her hands. She glanced round the glowing room as if some one might be watching her. Then, half ashamed, half joyful, she murmured “O Machine!” and raised the volume to her lips. Thrice she kissed it, thrice inclined her head, thrice she felt the delirium of acquiescence. Her ritual performed, she turned to page 1367, which gave the times of the departure of the air-ships from the island in the southern hemisphere, under whose soil she lived, to the island in the northern hemisphere, whereunder lived her son.

She thought, “I have not the time.”
She made the room dark and slept; she awoke and made the room light; she ate and exchanged ideas with her friends, and listened to music and attended lectures; she make the room dark and slept. Above her, beneath her, and around her, the Machine hummed eternally; she did not notice the noise, for she had been born with it in her ears. The earth, carrying her, hummed as it sped through silence, turning her now to the invisible sun, now to the invisible stars. She awoke and made the room light.

“Kuno!”

“I will not talk to you.” he answered, “until you come.”

“Have you been on the surface of the earth since we spoke last?”

His image faded.

Again she consulted the book. She became very nervous and lay back in her chair palpitating. Think of her as without teeth or hair. Presently she directed the chair to the wall, and pressed an unfamiliar button. The wall swung apart slowly. Through the opening she saw a tunnel that curved slightly, so that its goal was not visible. Should she go to see her son, here was the beginning of the journey.

Of course she knew all about the communication-system. There was nothing mysterious in it. She would summon a car and it would fly with her down the tunnel until it reached the lift that communicated with the air-ship station: the system had been in use for many, many years, long before the universal establishment of the Machine. And of course she had studied the civilization that had immediately preceded her own—the civilization that had mistaken the functions of the system, and had used it for bringing people to things, instead of for bringing things to people. Those funny old days, when men went for change of air instead of changing the air in their rooms! And yet—she was frightened of the tunnel: she had not seen it since her last child was born. It curved—but not quite as she remembered; it was brilliant—but not quite as brilliant as a lecturer had suggested. Vashti was seized with the terrors of direct experience. She shrank back into the room, and the wall closed up again.

“Kuno,” she said, “I cannot come to see you. I am not well.”

Immediately an enormous apparatus fell on to her out of the ceiling, a thermometer was automatically laid upon her heart. She lay powerless. Cool pads soothed her forehead. Kuno had telegraphed to her doctor.

So the human passions still blundered up and down in the Machine. Vashti drank the medicine that the doctor projected into her mouth, and the machinery retired into the ceiling. The voice of Kuno was heard asking how she felt.

“Better.” Then with irritation: “But why do you not come to me instead?”

“Because I cannot leave this place.”

“Why?”

“Because, any moment, something tremendous many happen.”

“Have you been on the surface of the earth yet?”

“Not yet.”

“Then what is it?”
“I will not tell you through the Machine.”

She resumed her life.

But she thought of Kuno as a baby, his birth, his removal to the public nurseries, her own visit to him there, his visits to her—visits which stopped when the Machine had assigned him a room on the other side of the earth. “Parents, duties of,” said the book of the Machine, “cease at the moment of birth.” True, but there was something special about Kuno—indeed there had been something special about all her children—and, after all, she must brave the journey if he desired it. And “something tremendous might happen”. What did that mean? The nonsense of a youthful man, no doubt, but she must go. Again she pressed the unfamiliar button, again the wall swung back, and she saw the tunnel that curves out of sight. Clasping the Book, she rose, tottered on to the platform, and summoned the car. Her room closed behind her: the journey to the northern hemisphere had begun.

Of course it was perfectly easy. The car approached and in it she found armchairs exactly like her own. When she signaled, it stopped, and she tottered into the lift. One other passenger was in the lift, the first fellow creature she had seen face to face for months. Few travelled in these days, for, thanks to the advance of science, the earth was exactly alike all over. Rapid intercourse, from which the previous civilization had hoped so much, had ended by defeating itself. What was the good of going to Peking when it was just like Shrewsbury? Why return to Shrewsbury when it would all be like Peking? Men seldom moved their bodies; all unrest was concentrated in the soul.

The air-ship service was a relic form the former age. It was kept up, because it was easier to keep it up than to stop it or to diminish it, but it now far exceeded the wants of the population. Vessel after vessel would rise from the vomitories of Rye or of Christchurch (I use the antique names), would sail into the crowded sky, and would draw up at the wharves of the south—empty. So nicely adjusted was the system, so independent of meteorology, that the sky, whether calm or cloudy, resembled a vast kaleidoscope whereon the same patterns periodically recurred. The ship on which Vashti sailed started now at sunset, now at dawn. But always, as it passed above Rhea, it would neighbour the ship that served between Helsingfors and the Brazils, and, every third time it surmounted the Alps, the fleet of Palermo would cross its track behind. Night and day, wind and storm, tide and earthquake, impeded man no longer. He had harnessed Leviathan. All the old literature, with its praise of Nature, and its fear of Nature, rang false as the prattle of a child.

Yet as Vashti saw the vast flank of the ship, stained with exposure to the outer air, her horror of direct experience returned. It was not quite like the air-ship in the cinematophote. For one thing it smelt—not strongly or unpleasantly, but it did smell, and with her eyes shut she should have known that a new thing was close to her. Then she had to walk to it from the lift, had to submit to glances from the other passengers. The man in front dropped his Book—no great matter, but it disquieted them all. In the rooms, if the Book was dropped, the floor raised it mechanically, but the gangway to the air-ship was not so prepared, and the sacred volume lay motionless. They stopped—the thing was unforeseen—and the man, instead of picking up his property, felt the muscles of his arm to see how they had failed him. Then some one actually said with direct utterance: “We shall be late”—and they trooped on board, Vashti treading on the pages as she did so.

Inside, her anxiety increased. The arrangements were old-fashioned and rough. There was even a female attendant, to whom she would have to announce her wants during the voyage. Of course a revolving platform ran the length of the boat, but she was expected to walk from it to her cabin. Some cabins were better than others, and she did not get the best. She thought the attendant had been unfair, and spasms of rage shook her. The glass valves had closed, she could not go back. She saw, at the end of the vestibule, the
lift in which she had ascended going quietly up and down, empty. Beneath those corridors of shining tiles were rooms, tier below tier, reaching far into the earth, and in each room there sat a human being, eating, or sleeping, or producing ideas. And buried deep in the hive was her own room. Vashti was afraid.

“O Machine!” she murmured, and caressed her Book, and was comforted.

Then the sides of the vestibule seemed to melt together, as do the passages that we see in dreams, the lift vanished, the Book that had been dropped slid to the left and vanished, polished tiles rushed by like a stream of water, there was a slight jar, and the air-ship, issuing from its tunnel, soared above the waters of a tropical ocean.

It was night. For a moment she saw the coast of Sumatra edged by the phosphorescence of waves, and crowned by lighthouses, still sending forth their disregarded beams. These also vanished, and only the stars distracted her. They were not motionless, but swayed to and fro above her head, thronging out of one sky-light into another, as if the universe and not the air-ship was careening. And, as often happens on clear nights, they seemed now to be in perspective, now on a plane; now piled tier beyond tier into the infinite heavens, now concealing infinity, a roof limiting for ever the visions of men. In either case they seemed intolerable. “Are we to travel in the dark?” called the passengers angrily, and the attendant, who had been careless, generated the light, and pulled down the blinds of pliable metal. When the air-ships had been built, the desire to look direct at things still lingered in the world. Hence the extraordinary number of skylights and windows, and the proportionate discomfort to those who were civilized and refined. Even in Vashti’s cabin one star peeped through a flaw in the blind, and after a few hours’ uneasy slumber, she was disturbed by an unfamiliar glow, which was the dawn.

Quick as the ship had sped westwards, the earth had rolled eastwards quicker still, and had dragged back Vashti and her companions towards the sun. Science could prolong the night, but only for a little, and those high hopes of neutralizing the earth’s diurnal revolution had passed, together with hopes that were possibly higher. To “keep pace with the sun,” or even to outstrip it, had been the aim of the civilization preceding this. Racing aeroplanes had been built for the purpose, capable of enormous speed, and steered by the greatest intellects of the epoch. Round the globe they went, round and round, westward, westward, round and round, amidst humanity’s applause. In vain. The globe went eastward quicker still, horrible accidents occurred, and the Committee of the Machine, at the time rising into prominence, declared the pursuit illegal, unmechanical, and punishable by Homelessness.

Of Homelessness more will be said later.

Doubtless the Committee was right. Yet the attempt to “defeat the sun” aroused the last common interest that our race experienced about the heavenly bodies, or indeed about anything. It was the last time that men were compacted by thinking of a power outside the world. The sun had conquered, yet it was the end of his spiritual dominion. Dawn, midday, twilight, the zodiacal path, touched neither men’s lives not their hearts, and science retreated into the ground, to concentrate herself upon problems that she was certain of solving.

So when Vashti found her cabin invaded by a rosy finger of light, she was annoyed, and tried to adjust the blind. But the blind flew up altogether, and she saw through the skylight small pink clouds, swaying against a background of blue, and as the sun crept higher, its radiance entered direct, brimming down the wall, like a golden sea. It rose and fell with the air-ship’s motion, just as waves rise and fall, but it advanced steadily, as a tide advances. Unless she was careful, it would strike her face. A spasm of horror shook her and she rang for the attendant. The attendant too was horrified, but she could do nothing; it was not her place to mend the blind. She could only suggest that the lady should change her cabin, which she accordingly prepared to do.
People were almost exactly alike all over the world, but the attendant of the air-ship, perhaps owing to her exceptional duties, had grown a little out of the common. She had often to address passengers with direct speech, and this had given her a certain roughness and originality of manner. When Vashti served away form the sunbeams with a cry, she behaved barbarically—she put out her hand to steady her.

“How dare you!” exclaimed the passenger. “You forget yourself!”

The woman was confused, and apologized for not having let her fall. People never touched one another. The custom had become obsolete, owing to the Machine.

“Where are we now?” asked Vashti haughtily.

“We are over Asia,” said the attendant, anxious to be polite.

“Asia?”

“You must excuse my common way of speaking. I have got into the habit of calling places over which I pass by their unmechanical names.”

“Oh, I remember Asia. The Mongols came from it.”

“Beneath us, in the open air, stood a city that was once called Simla.”

“Have you ever heard of the Mongols and of the Brisbane school?”

“No.”

“I have forgotten its name.”

“Brisbane also stood in the open air.”

“No.”

“She pushed back a metal blind. The main chain of the Himalayas was revealed. “They were once called the Roof of the World, those mountains.”

“You must remember that, before the dawn of civilization, they seemed to be an impenetrable wall that touched the stars. It was supposed that no one but the gods could exist above their summits. How we have advanced, thanks to the Machine!”

“How we have advanced, thanks to the Machine!” said Vashti.

“How we have advanced, thanks to the Machine!” echoed the passenger who had dropped his Book the night before, and who was standing in the passage.

“And that white stuff in the cracks?—what is it?”

“I have forgotten its name.”

“Cover the window, please. These mountains give me no ideas.”

The northern aspect of the Himalayas was in deep shadow: on the Indian slope the sun had just prevailed. The forests had been destroyed during the literature epoch for the purpose of making newspaper-pulp, but the snows were awakening to their morning glory, and clouds still hung on the breasts of Kinchinjunga. In the plain were seen the ruins of cities, with diminished rivers creeping by their walls, and by the sides of these were sometimes the signs of vomitories, marking the cities of to day. Over the whole prospect air-ships rushed, crossing the inter-crossing with incredible aplomb, and rising nonchalantly when they desired to escape the perturbations of the lower atmosphere and to traverse the Roof of the World.
“We have indeed advanced, thanks to the Machine,” repeated the attendant, and hid the Himalayas behind a metal blind.

The day dragged wearily forward. The passengers sat each in his cabin, avoiding one another with an almost physical repulsion and longing to be once more under the surface of the earth. There were eight or ten of them, mostly young males, sent out from the public nurseries to inhabit the rooms of those who had died in various parts of the earth. The man who had dropped his Book was on the homeward journey. He had been sent to Sumatra for the purpose of propagating the race. Vashti alone was travelling by her private will.

At midday she took a second glance at the earth. The air-ship was crossing another range of mountains, but she could see little, owing to clouds. Masses of black rock hovered below her, and merged indistinctly into grey. Their shapes were fantastic; one of them resembled a prostrate man.

“No ideas here,” murmured Vashti, and hid the Caucasus behind a metal blind.

In the evening she looked again. They were crossing a golden sea, in which lay many small islands and one peninsula. She repeated, “No ideas here,” and hid Greece behind a metal blind.

II THE MENDING APPARATUS

By a vestibule, by a lift, by a tubular railway, by a platform, by a sliding door—by reversing all the steps of her departure did Vashti arrive at her son’s room, which exactly resembled her own. She might well declare that the visit was superfluous. The buttons, the knobs, the reading-desk with the Book, the temperature, the atmosphere, the illumination—all were exactly the same. And if Kuno himself, flesh of her flesh, stood close beside her at last, what profit was there in that? She was too well-bred to shake him by the hand.

Averting her eyes, she spoke as follows:

“Here I am. I have had the most terrible journey and greatly retarded the development of my soul. It is not worth it, Kuno, it is not worth it. My time is too precious. The sunlight almost touched me, and I have met with the rudest people. I can only stop a few minutes. Say what you want to say, and then I must return.”

“I have been threatened with Homelessness,” said Kuno.

She looked at him now.

“I have been threatened with Homelessness, and I could not tell you such a thing through the Machine.”

Homelessness means death. The victim is exposed to the air, which kills him.

“I have been outside since I spoke to you last. The tremendous thing has happened, and they have discovered me.”

“But why shouldn’t you go outside?” she exclaimed, “It is perfectly legal, perfectly mechanical, to visit the surface of the earth. I have lately been to a lecture on the sea; there is no objection to that; one simply summons a respirator and gets an Egression-permit. It is not the kind of thing that spiritually minded people do, and I begged you not to do it, but there is no legal objection to it.”

“I did not get an Egression-permit.”

“Then how did you get out?”
“I found out a way of my own.”

The phrase conveyed no meaning to her, and he had to repeat it.

“A way of your own?” she whispered. “But that would be wrong.”

“Why?”

The question shocked her beyond measure.

“You are beginning to worship the Machine,” he said coldly.

“You think it irreligious of me to have found out a way of my own. It was just what the Committee thought, when they threatened me with Homelessness.”

At this she grew angry. “I worship nothing!” she cried. “I am most advanced. I don’t think you irreligious, for there is no such thing as religion left. All the fear and the superstition that existed once have been destroyed by the Machine. I only meant that to find out a way of your own was—Besides, there is no new way out.”

“So it is always supposed.”

“Except through the vomitories, for which one must have an Egression-permit, it is impossible to get out. The Book says so.”

“Well, the Book’s wrong, for I have been out on my feet.”

For Kuno was possessed of a certain physical strength.

By these days it was a demerit to be muscular. Each infant was examined at birth, and all who promised undue strength were destroyed. Humanitarians may protest, but it would have been no true kindness to let an athlete live; he would never have been happy in that state of life to which the Machine had called him; he would have yearned for trees to climb, rivers to bathe in, meadows and hills against which he might measure his body. Man must be adapted to his surroundings, must he not? In the dawn of the world our weakly must be exposed on Mount Taygetus, in its twilight our strong will suffer euthanasia, that the Machine may progress, that the Machine may progress, that the Machine may progress eternally.

“You know that we have lost the sense of space. We say ‘space is annihilated,’ but we have annihilated not space, but the sense thereof. We have lost a part of ourselves. I determined to recover it, and I began by walking up and down the platform of the railway outside my room. Up and down, until I was tired, and so did recapture the meaning of ‘Near’ and ‘Far’. ‘Near’ is a place to which I can get quickly on my feet, not a place to which the train or the air-ship will take me quickly. ‘Far’ is a place to which I cannot get quickly on my feet; the vomitory is ‘far,’ though I could be there in thirty-eight seconds by summoning the train. Man is the measure. That was my first lesson. Man’s feet are the measure for distance, his hands are the measure for ownership, his body is the measure for all that is lovable and desirable and strong. Then I went further: it was then that I called to you for the first time, and you would not come.

“This city, as you know, is built deep beneath the surface of the earth, with only the vomitories protruding. Having paced the platform outside my own room, I took the lift to the next platform and paced that also, and so with each in turn, until I came to the topmost, above which begins the earth. All the platforms were exactly alike, and all that I gained by visiting them was to develop my sense of space and my muscles. I think I should have been content with this—it is not a little thing—but as I walked and brooded, it occurred to me that our cities had been built in the days when men still breathed the outer air,
and that there had been ventilation shafts for the workmen. I could think of nothing but these ventilation
shafts. Had they been destroyed by all the food-tubes and medicine-tubes and music-tubes that the Machine
has evolved lately? Or did traces of them remain? One thing was certain. If I came upon them anywhere, it
would be in the railway-tunnels of the topmost storey. Everywhere else, all space was accounted for.

"I am telling my story quickly, but don't think that I was not a coward or that your answers never
depressed me. It is not the proper thing, it is not mechanical, it is not decent to walk along a railway-tunnel.
I did not fear that I might tread upon a live rail and be killed. I feared something far more intangible—
doing what was not contemplated by the Machine. Then I said to myself, 'Man is the measure,' and I went,
and after many visits I found an opening.

"The tunnels, of course, were lighted. Everything is light, artificial light; darkness is the exception. So
when I saw a black gap in the tiles, I knew that it was an exception, and rejoiced. I put in my arm—I could
put in no more at first—and waved it round and round in ecstasy. I loosened another tile, and put in my
head, and shouted into the darkness: 'I am coming, I shall do it yet,' and my voice reverberated down
endless passages. I seemed to hear the spirits of those dead workmen who had returned each evening to the
starlight and to their wives, and all the generations who had lived in the open air called back to me, 'You
will do it yet, you are coming.'"

He paused, and, absurd as he was, his last words moved her.

For Kuno had lately asked to be a father, and his request had been refused by the Committee. His was
not a type that the Machine desired to hand on.

"Then a train passed. It brushed by me, but I thrust my head and arms into the hole. I had done enough
for one day, so I crawled back to the platform, went down in the lift, and summoned my bed. Ah what
dreams! And again I called you, and again you refused."

She shook her head and said:

"Don't. Don't talk of these terrible things. You make me miserable. You are throwing civilization
away."

"But I had got back the sense of space and a man cannot rest then. I determined to get in at the hole and
climb the shaft. And so I exercised my arms. Day after day I went through ridiculous movements, until my
flesh ached, and I could hang by my hands and hold the pillow of my bed outstretched for many minutes.
Then I summoned a respirator, and started.

"It was easy at first. The mortar had somehow rotted, and I soon pushed some more tiles in, and
clambered after them into the darkness, and the spirits of the dead comforted me. I don't know what I mean
by that. I just say what I felt. I felt, for the first time, that a protest had been lodged against corruption, and
that even as the dead were comforting me, so I was comforting the unborn. I felt that humanity existed, and
that it existed without clothes. How can I possibly explain this? It was naked, humanity seemed naked, and
all these tubes and buttons and machineries neither came into the world with us, nor will they follow us out,
nor do they matter supremely while we are here. Had I been strong, I would have torn off every garment I
had, and gone out into the outer air unswaddled. But this is not for me, nor perhaps for my generation. I
climbed with my respirator and my hygienic clothes and my dietetic tabloids! Better thus than not at all.

"There was a ladder, made of some primeval metal. The light from the railway fell upon its lowest
rungs, and I saw that it led straight upwards out of the rubble at the bottom of the shaft. Perhaps our
ancestors ran up and down it a dozen times daily, in their building. As I climbed, the rough edges cut
through my gloves so that my hands bled. The light helped me for a little, and then came darkness and,
worse still, silence which pierced my ears like a sword. The Machine hums! Did you know that? Its hum penetrates our blood, and may even guide our thoughts. Who knows! I was getting beyond its power. Then I thought: ‘This silence means that I am doing wrong.’ But I heard voices in the silence, and again they strengthened me.” He laughed. “I had need of them. The next moment I cracked my head against something.”

She sighed.

“I had reached one of those pneumatic stoppers that defend us from the outer air. You may have noticed them no the air-ship. Pitch dark, my feet on the rungs of an invisible ladder, my hands cut; I cannot explain how I lived through this part, but the voices till comforted me, and I felt for fastenings. The stopper, I suppose, was about eight feet across. I passed my hand over it as far as I could reach. It was perfectly smooth. I felt it almost to the centre. Not quite to the centre, for my arm was too short. Then the voice said: ‘Jump. It is worth it. There may be a handle in the centre, and you may catch hold of it and so come to us your own way. And if there is no handle, so that you may fall and are dashed to pieces—it is till worth it: you will still come to us your own way.’ So I jumped. There was a handle, and—”

He paused. Tears gathered in his mother’s eyes. She knew that he was fated. If he did not die today he would die tomorrow. There was not room for such a person in the world. And with her pity disgust mingled. She was ashamed at having borne such a son, she who had always been so respectable and so full of ideas. Was he really the little boy to whom she had taught the use of his stops and buttons, and to whom she had given his first lessons in the Book? The very hair that disfigured his lip showed that he was reverting to some savage type. On atavism the Machine can have no mercy.

“There was a handle, and I did catch it. I hung tranced over the darkness and heard the hum of these workings as the last whisper in a dying dream. All the things I had cared about and all the people I had spoken to through tubes appeared infinitely little. Meanwhile the handle revolved. My weight had set something in motion and I span slowly, and then—

“I cannot describe it. I was lying with my face to the sunshine. Blood poured from my nose and ears and I heard a tremendous roaring. The stopper, with me clinging to it, had simply been blown out of the earth, and the air that we make down here was escaping through the vent into the air above. It burst up like a fountain. I crawled back to it—for the upper air hurts—and, as it were, I took great sips from the edge. My respirator had flown goodness knows here, my clothes were torn. I just lay with my lips close to the hole, and I sipped until the bleeding stopped. You can imagine nothing so curious. This hollow in the grass—I will speak of it in a minute,—the sun shining into it, not brilliantly but through marbled clouds,—the peace, the nonchalance, the sense of space, and, brushing my cheek, the roaring fountain of our artificial air! Soon I spied my respirator, bobbing up and down in the current high above my head, and higher still were many air-ships. But no one ever looks out of air-ships, and in any case they could not have picked me up. There I was, stranded. The sun shone a little way down the shaft, and revealed the topmost rung of the ladder, but it was hopeless trying to reach it. I should either have been tossed up again by the escape, or else have fallen in, and died. I could only lie on the grass, sipping and sipping, and from time to time glancing around me.

“I knew that I was in Wessex, for I had taken care to go to a lecture on the subject before starting. Wessex lies above the room in which we are talking now. It was once an important state. Its kings held all the southern coast form the Andredswald to Cornwall, while the Wansdyke protected them on the north, running over the high ground. The lecturer was only concerned with the rise of Wessex, so I do not know how long it remained an international power, nor would the knowledge have assisted me. To tell the truth I could do nothing but laugh, during this part. There was I, with a pneumatic stopper by my side and a
respirator bobbing over my head, imprisoned, all three of us, in a grass-grown hollow that was edged with fern.”

Then he grew grave again.

“Lucky for me that it was a hollow. For the air began to fall back into it and to fill it as water fills a bowl. I could crawl about. Presently I stood. I breathed a mixture, in which the air that hurts predominated whenever I tried to climb the sides. This was not so bad. I had not lost my tabloids and remained ridiculously cheerful, and as for the Machine, I forgot about it altogether. My one aim now was to get to the top, where the ferns were, and to view whatever objects lay beyond.

“I rushed the slope. The new air was still too bitter for me and I came rolling back, after a momentary vision of something grey. The sun grew very feeble, and I remembered that he was in Scorpio—I had been to a lecture on that too. If the sun is in Scorpio, and you are in Wessex, it means that you must be as quick as you can, or it will get too dark. (This is the first bit of useful information I have ever got from a lecture, and I expect it will be the last.) It made me try frantically to breathe the new air, and to advance as far as I dared out of my pond. The hollow filled so slowly. At times I thought that the fountain played with less vigour. My respirator seemed to dance nearer the earth; the roar was decreasing.”

He broke off.

“I don’t think this is interesting you. The rest will interest you even less. There are no ideas in it, and I wish that I had not troubled you to come. We are too different, mother.”

She told him to continue.

“It was evening before I climbed the bank. The sun had very nearly slipped out of the sky by this time, and I could not get a good view. You, who have just crossed the Roof of the World, will not want to hear an account of the little hills that I saw—low colourless hills. But to me they were living and the turf that covered them was a skin, under which their muscles rippled, and I felt that those hills had called with incalculable force to men in the past, and that men had loved them. Now they sleep—perhaps for ever. They commune with humanity in dreams. Happy the man, happy the woman, who awakes the hills of Wessex. For though they sleep, they will never die.”

His voice rose passionately.

“Cannot you see, cannot all you lecturers see, that it is we that are dying, and that down here the only thing that really lives in the Machine? We created the Machine, to do our will, but we cannot make it do our will now. It has robbed us of the sense of space and of the sense of touch, it has blurred every human relation and narrowed down love to a carnal act, it has paralysed our bodies and our wills, and now it compels us to worship it. The Machine develops—but not our lives. The Machine proceeds—but not to our goal. We only exist as the blood corpuscles that course through its arteries, and if it could work without us, it would let us die. Oh, I have no remedy—or, at least, only one—to tell men again and again that I have seen the hills of Wessex as Alfrid saw them when he overthrew the Danes.

“So the sun set. I forgot to mention that a belt of mist lay between my hill and other hills, and that it was the colour of pearl.”

He broke off for the second time.

“Go on,” said his mother wearily.

He shook his head.
“Go on. Nothing that you say can distress me now. I am hardened.”

“I had meant to tell you the rest, but I cannot: I know that I cannot: good-bye.”

Vashti stood irresolute. All her nerves were tingling with his blasphemies. But she was also inquisitive.

“This is unfair,” she complained. “You have called me across the world to hear your story, and hear it I will. Tell me—as briefly as possible, for this is a disastrous waste of time—tell me how you returned to civilization.”

“Oh—that!” he said, starting. “You would like to hear about civilization. Certainly. Had I got to where my respirator fell down?”

“No—but I understand everything now. You put on your respirator, and managed to walk along the surface of the earth to a vomitory, and there your conduct was reported to the Central Committee.”

“By no means.”

He passed his hand over his forehead, as if dispelling some strong impression. Then, resuming his narrative, he warmed to it again.

“My respirator fell about sunset. I had mentioned that the fountain seemed feebler, had I not?”

“Yes.”

“About sunset, it let the respirator fall. As I said, I had entirely forgotten about the Machine, and I paid no great attention at the time, being occupied with other things. I had my pool of air, into which I could dip when the outer keenness became intolerable, and which would possibly remain for days, provided that no wind sprang up to disperse it. Not until it was too late did I realize what the stoppage of the escape implied. You see—the gap in the tunnel had been mended; the Mending Apparatus; the Mending Apparatus, was after me.

“One other warning I had, but I neglected it. The sky at night was clearer than it had been in the day, and the moon, which was about half the sky behind the sun, shone into the dell at moments quite brightly. I was in my usual place—on the boundary between the two atmospheres—when I thought I saw something dark move across the bottom of the dell, and vanish into the shaft. In my folly, I ran down. I bent over and listened, and I thought I heard a faint scraping noise in the depths.

“At this—but it was too late—I took alarm. I determined to put on my respirator and to walk right out of the dell. But my respirator had gone. I knew exactly where it had fallen—between the stopper and the aperture—and I could even feel the mark that it had made in the turf. It had gone, and I realized that something evil was at work, and I had better escape to the other air, and, if I must die, die running towards the cloud that had been the colour of a pearl. I never started. Out of the shaft—it is too horrible. A worm, a long white worm, had crawled out of the shaft and gliding over the moonlit grass.

“I screamed. I did everything that I should not have done, I stamped upon the creature instead of flying from it, and it at once curled round the ankle. Then we fought. The worm let me run all over the dell, but edged up my leg as I ran. ‘Help!’ I cried. (That part is too awful. It belongs to the part that you will never know.) ‘Help!’ I cried. (Why cannot we suffer in silence?) ‘Help!’ I cried. When my feet were wound together, I fell, I was dragged away from the dear ferns and the living hills, and past the great metal stopper (I can tell you this part), and I thought it might save me again if I caught hold of the handle. It also was enwrapped, it also. Oh, the whole dell was full of the things. They were searching it in all directions, they
were denuding it, and the white snouts of others peeped out of the hole, ready if needed. Everything that could be moved they brought—brushwood, bundles of fern, everything, and down we all went intertwined into hell. The last things that I saw, ere the stopper closed after us, were certain stars, and I felt that a man of my sort lived in the sky. For I did fight, I fought till the very end, and it was only my head hitting against the ladder that quieted me. I woke up in this room. The worms had vanished. I was surrounded by artificial air, artificial light, artificial peace, and my friends were calling to me down speaking-tubes to know whether I had come across any new ideas lately."

Here his story ended. Discussion of it was impossible, and Vashti turned to go.

"It will end in Homelessness," she said quietly.

"I wish it would," retorted Kuno.

"The Machine has been most merciful."

"I prefer the mercy of God."

"By that superstitious phrase, do you mean that you could live in the outer air?"

"Yes."

"Have you ever seen, round the vomitories, the bones of those who were extruded after the Great Rebellion?"

"Yes."

"They were left where they perished for our edification. A few crawled away, but they perished, too—who can doubt it? And so with the Homeless of our own day. The surface of the earth supports life no longer."

"Indeed."

"Ferns and a little grass may survive, but all higher forms have perished. Has any air-ship detected them?"

"No."

"Has any lecturer dealt with them?"

"No."

"Then why this obstinacy?"

"Because I have seen them," he exploded.

"Seen what?"

"Because I have seen her in the twilight—because she came to my help when I called—because she, too, was entangled by the worms, and, luckier than I, was killed by one of them piercing her throat."

He was mad. Vashti departed, nor, in the troubles that followed, did she ever see his face again.
During the years that followed Kuno’s escapade, two important developments took place in the Machine. On the surface they were revolutionary, but in either case men’s minds had been prepared beforehand, and they did but express tendencies that were latent already.

The first of these was the abolition of respirator.

Advanced thinkers, like Vashti, had always held it foolish to visit the surface of the earth. Air-ships might be necessary, but what was the good of going out for mere curiosity and crawling along for a mile or two in a terrestrial motor? The habit was vulgar and perhaps faintly improper: it was unproductive of ideas, and had no connection with the habits that really mattered. So respirators were abolished, and with them, of course, the terrestrial motors, and except for a few lecturers, who complained that they were debarred access to their subject-matter, the development was accepted quietly. Those who still wanted to know what the earth was like had after all only to listen to some gramophone, or to look into some cinematophote. And even the lecturers acquiesced when they found that a lecture on the sea was none the less stimulating when compiled out of other lectures that had already been delivered on the same subject. “Beware of first-hand ideas!” exclaimed one of the most advanced of them. “First-hand ideas do not really exist. They are but the physical impressions produced by live and fear, and on this gross foundation who could erect a philosophy? Let your ideas be second-hand, and if possible tenth-hand, for then they will be far removed from that disturbing element—direct observation. Do not learn anything about this subject of mine—the French Revolution. Learn instead what I think that Enicharmon thought Urizen thought Gutch thought Ho-Yung thought Chi-Bo-Sing thought LafcadioHearn thought Carlyle thought Mirabeau said about the French Revolution. Through the medium of these ten great minds, the blood that was shed at Paris and the windows that were broken at Versailles will be clarified to an idea which you may employ most profitably in your daily lives. But be sure that the intermediates are many and varied, for in history one authority exists to counteract another. Urizen must counteract the scepticism of Ho-Yung and Enicharmon, I must myself counteract the impetuosity of Gutch. You who listen to me are in a better position to judge about the French Revolution than I am. Your descendants will be even in a better position than you, for they will learn what you think I think, and yet another intermediate will be added to the chain. And in time”—his voice rose—“there will come a generation that had got beyond facts, beyond impressions, a generation absolutely colourless, a generation

seraphically free
From taint of personality

which will see the French Revolution not as it happened, nor as they would like it to have happened, but as it would have happened, had it taken place in the days of the Machine.”

Tremendous applause greeted this lecture, which did but voice a feeling already latent in the minds of men—a feeling that terrestrial facts must be ignored, and that the abolition of respirators was a positive gain. It was even suggested that air-ships should be abolished too. This was not done, because air-ships had somehow worked themselves into the Machine’s system. But year by year they were used less, and mentioned less by thoughtful men.

The second great development was the re-establishment of religion.

This, too, had been voiced in the celebrated lecture. No one could mistake the reverent tone in which the peroration had concluded, and it awakened a responsive echo in the heart of each. Those who had long worshipped silently, now began to talk. They described the strange feeling of peace that came over them when they handled the Book of the Machine, the pleasure that it was to repeat certain numerals out of it,
however little meaning those numerals conveyed to the outward ear, the ecstasy of touching a button, however unimportant, or of ringing an electric bell, however superfluously.

“The Machine,” they exclaimed, “feeds us and clothes us and houses us; through it we speak to one another, through it we see one another, in it we have our being. The Machine is the friend of ideas and the enemy of superstition: the Machine is omnipotent, eternal; blessed is the Machine.” And before long this allocution was printed on the first page of the Book, and in subsequent editions the ritual swelled into a complicated system of praise and prayer. The word ‘religion’ was sedulously avoided, and in theory the Machine was still the creation and the implement of man, but in practice all, save a few retrogrades, worshipped it as divine. Nor was it worshipped in unity. One believer would be chiefly impressed by the blue optic plates, through which he saw other believers; another by the mending apparatus, which sinful Kuno had compared to worms; another by the lifts, another by the Book. And each would pray to this or to that, and ask it to intercede for him with the Machine as a whole. Persecution—that also was present. It did not break out, for reasons that will be set forward shortly. But it was latent, and all who did not accept the minimum known as ‘undenominational Mechanism’ lived in danger of Homelessness, which means death, as we know.

To attribute these two great developments to the Central Committee, is to take a very narrow view of civilization. The Central Committee announced the developments, it is true, but they were no more the cause of them than were the kings of the imperialistic period the cause of war. Rather did they yield to some invincible pressure, which came no one knew whither, and which, when gratified, was succeeded by some new pressure equally invincible. To such a state of affairs it is convenient to give the name of progress. No one confessed the Machine was out of hand. Year by year it was served with increased efficiency and decreased intelligence. The better a man knew his own duties upon it, the less he understood the duties of his neighbour, and in all the world there was not one who understood the monster as a whole. Those master brains had perished. They had left full directions, it is true, and their successors had each of them mastered a portion of those directions. But Humanity, in its desire for comfort, had over-reached itself. It had exploited the riches of nature too far. Quietly and complacently, it was sinking into decadence, and progress had come to mean the progress of the Machine.

As for Vashti, her life went peacefully forward until the final disaster. She made her room dark and slept; she awoke and made the room light. She lectured and attended lectures. She exchanged ideas with her innumerable friends and believed she was growing more spiritual. At times a friend was granted Euthanasia, and left his or her room for the homelessness that is beyond all human conception. Vashti did not much mind. After an unsuccessful lecture, she would sometimes ask for Euthanasia herself. But the death-rate was not permitted to exceed the birth-rate, and the Machine had hitherto refused it to her.

The troubles began quietly, long before she was conscious of them.

One day she was astonished at receiving a message from her son. They never communicated, having nothing in common, and she had only heard indirectly that he was still alive, and had been transferred from the northern hemisphere, where he had behaved so mischievously, to the southern—indeed, to a room not far from her own.

“Does he want me to visit him?” she thought. “Never again, never. And I have not the time.”

No, it was madness of another kind.

He refused to visualize his face upon the blue plate, and speaking out of the darkness with solemnity said:
“The Machine stops.”
“What do you say?”
“The Machine is stopping, I know it, I know the signs.”

She burst into a peal of laughter. He heard her and was angry, and they spoke no more.

“Can you imagine anything more absurd?” she cried to a friend. “A man who was my son believes that the Machine is stopping. It would be impious if it was not mad.”

“The Machine is stopping?” her friend replied. “What does that mean? The phrase conveys nothing to me.”

“Nor to me.”

“He does not refer, I suppose, to the trouble there has been lately with the music?”

“Oh no, of course not. Let us talk about music.”

“Have you complained to the authorities?”

“Yes, and they say it wants mending, and referred me to the Committee of the Mending Apparatus. I complained of those curious gasping sighs that disfigure the symphonies of the Brisbane school. They sound like some one in pain. The Committee of the Mending Apparatus say that it shall be remedied shortly.”

Obscurely worried, she resumed her life. For one thing, the defect in the music irritated her. For another thing, she could not forget Kuno’s speech. If he had known that the music was out of repair—he could not know it, for he detested music—if he had known that it was wrong, “the Machine stops” was exactly the venomous sort of remark he would have made. Of course he had made it at a venture, but the coincidence annoyed her, and she spoke with some petulance to the Committee of the Mending Apparatus.

They replied, as before, that the defect would be set right shortly.

“Shortly! At once!” she retorted. “Why should I be worried by imperfect music? Things are always put right at once. If you do not mend it at once, I shall complain to the Central Committee.”

“No personal complaints are received by the Central Committee,” the Committee of the Mending Apparatus replied.

“Through whom am I to make my complaint, then?”

“Through us.”

“I complain then.”

“Your complaint shall be forwarded in its turn.”

“Have others complained?”

This question was unmechanical, and the Committee of the Mending Apparatus refused to answer it.

“It is too bad!” she exclaimed to another of her friends.
“There never was such an unfortunate woman as myself. I can never be sure of my music now. It gets worse and worse each time I summon it.”

“What is it?”

“I do not know whether it is inside my head, or inside the wall.”

“Complain, in either case.”

“I have complained, and my complaint will be forwarded in its turn to the Central Committee.”

Time passed, and they resented the defects no longer. The defects had not been remedied, but the human tissues in that latter day had become so subservient, that they readily adapted themselves to every caprice of the Machine. The sigh at the crises of the Brisbane symphony no longer irritated Vashti; she accepted it as part of the melody. The jarring noise, whether in the head or in the wall, was no longer resented by her friend. And so with the mouldy artificial fruit, so with the bath water that began to stink, so with the defective rhymes that the poetry machine had taken to emit. All were bitterly complained of at first, and then acquiesced in and forgotten. Things went from bad to worse unchallenged.

It was otherwise with the failure of the sleeping apparatus. That was a more serious stoppage. There came a day when over the whole world—in Sumatra, in Wessex, in the innumerable cities of Courland and Brazil—the beds, when summoned by their tired owners, failed to appear. It may seem a ludicrous matter, but from it we may date the collapse of humanity. The Committee responsible for the failure was assailed by complainants, whom it referred, as usual, to the Committee of the Mending Apparatus, who in its turn assured them that their complaints would be forwarded to the Central Committee. But the discontent grew, for mankind was not yet sufficiently adaptable to do without sleeping.

“Some one is meddling with the Machine—” they began.

“Some one is trying to make himself king, to reintroduce the personal element.”

“Punish that man with Homelessness.”

“To the rescue! Avenge the Machine! Avenge the Machine!”

“War! Kill the man!”

But the Committee of the Mending Apparatus now came forward, and allayed the panic with well-chosen words. It confessed that the Mending Apparatus was itself in need of repair.

The effect of this frank confession was admirable.

“Of course,” said a famous lecturer—he of the French Revolution, who gilded each new decay with splendour—“of course we shall not press our complaints now. The Mending Apparatus has treated us so well in the past that we all sympathize with it, and will wait patiently for its recovery. In its own good time it will resume its duties. Meanwhile let us do without our beds, our tabloids, our other little wants. Such, I feel sure, would be the wish of the Machine.”

Thousands of miles away his audience applauded. The Machine still linked them. Under the seas, beneath the roots of the mountains, ran the wires through which they saw and heard, the enormous eyes and ears that were their heritage, and the hum of many workings clothed their thoughts in one garment of subserviency. Only the old and the sick remained ungrateful, for it was rumoured that Euthanasia, too, was out of order, and that pain had reappeared among men.
It became difficult to read. A blight entered the atmosphere and dulled its luminosity. At times Vashti could scarcely see across her room. The air, too, was foul. Loud were the complaints, impotent the remedies, heroic the tone of the lecturer as he cried: “Courage! courage! What matter so long as the Machine goes on? To it the darkness and the light are one.” And though things improved again after a time, the old brilliancy was never recaptured, and humanity never recovered from its entrance into twilight.

There was an hysterical talk of ‘measures,’ of ‘provisional dictatorship,’ and the inhabitants of Sumatra were asked to familiarize themselves with the workings of the central power station, the said power station being situated in France. But for the most part panic reigned, and men spent their strength praying to their Books, tangible proofs of the Machine’s omnipotence. There were gradations of terror—at times came rumours of hope—the Mending Apparatus was almost mended—the enemies of the Machine had been got under—new ‘nerve-centres’ were evolving which would do the work even more magnificently than before. But there came a day when, without the slightest warning, without any previous hint of feebleness, the entire communication-system broke down, all over the world, and the world, as they understood it, ended.

Vashti was lecturing at the time and her earlier remarks had been punctuated with applause. As she proceeded the audience became silent, and at the conclusion there was no sound. Somewhat displeased, she called to a friend who was a specialist in sympathy. No sound: doubtless the friend was sleeping. And so with the next friend whom she tried to summon, and so with the next, until she remembered Kuno’s cryptic remark, “The Machine stops.”

The phrase still conveyed nothing. If Eternity was stopping it would of course be set going shortly.

For example, there was still a little light and air—the atmosphere had improved a few hours previously. There was still the Book, and while there was the Book there was security.

Then she broke down, for with the cessation of activity came an unexpected terror—silence.

She had never known silence, and the coming of it nearly killed her—it did kill many thousands of people outright. Ever since her birth she had been surrounded by the steady hum. It was to the ear what artificial air was to the lungs, and agonizing pains shot across her head. And scarcely knowing what she did, she stumbled forward and pressed the unfamiliar button, the one that opened the door of her cell.

Now the door of the cell worked on a simple hinge of its own. It was not connected with the central power station, dying far away in France. It opened, rousing immoderate hopes in Vashti, for she thought that the Machine had been mended. It opened, and she saw the dim tunnel that curved far away towards freedom. One look, and then she shrank back. For the tunnel was full of people—she was almost the last in that city to have taken alarm.

People at any time repelled her, and these were nightmares from her worst dreams. People were crawling about, people were screaming, whimpering, gasping for breath, touching each other, vanishing in the dark, and ever and anon being pushed off the platform on to the live rail. Some were fighting round the electric bells, trying to summon trains which could not be summoned. Others were yelling for Euthanasia or for respirators, or blaspheming the Machine. Others stood at the doors of their cells fearing, like herself, either to stop in them or to leave them. And behind all the uproar was silence—the silence which is the voice of the earth and of the generations who have gone.

No—it was worse than solitude. She closed the door again and sat down to wait for the end. The disintegration went on, accompanied by horrible cracks and rumbling. The valves that restrained the Medical Apparatus must have weakened, for it ruptured and hung hideously from the ceiling. The floor heaved and fell and flung her from the chair. A tube oozed towards her serpent fashion. And at last the final horror approached—light began to ebb, and she knew that civilization’s long day was closing.
She whirled around, praying to be saved from this, at any rate, kissing the Book, pressing button after button. The uproar outside was increasing, and even penetrated the wall. Slowly the brilliancy of her cell was dimmed, the reflections faded from the metal switches. Now she could not see the reading-stand, now not the Book, though she held it in her hand. Light followed the flight of sound, air was following light, and the original void returned to the cavern from which it has so long been excluded. Vashti continued to whirl, like the devotees of an earlier religion, screaming, praying, striking at the buttons with bleeding hands.

It was thus that she opened her prison and escaped—escaped in the spirit: at least so it seems to me, ere my meditation closes. That she escapes in the body—I cannot perceive that. She struck, by chance, the switch that released the door, and the rush of foul air on her skin, the loud throbbing whispers in her ears, told her that she was facing the tunnel again, and that tremendous platform on which she had seen men fighting. They were not fighting now. Only the whispers remained, and the little whimpering groans. They were dying by hundreds out in the dark.

She burst into tears.

Tears answered her.

They wept for humanity, those two, not for themselves. They could not bear that this should be the end. Ere silence was completed their hearts were opened, and they knew what had been important on the earth. Man, the flower of all flesh, the noblest of all creatures visible, man who had once made god in his image, and had mirrored his strength on the constellations, beautiful naked man was dying, strangled in the garments that he had woven. Century after century had he toiled, and here was his reward. Truly the garment had seemed heavenly at first, shot with colours of culture, sewn with the threads of self-denial. And heavenly it had been so long as man could shed it at will and live by the essence that is his soul, and the essence, equally divine, that is his body. The sin against the body—it was for that they wept in chief; the centuries of wrong against the muscles and the nerves, and those five portals by which we can alone apprehend—glozing it over with talk of evolution, until the body was white pap, the home of ideas as colourless, last sloshy stirrings of a spirit that had grasped the stars.

“Where are you?” she sobbed.

His voice in the darkness said, “Here.”

“Is there any hope, Kuno?”

“None for us.”

“Where are you?”

She crawled over the bodies of the dead. His blood spurted over her hands.

“Quicker,” he gasped, “I am dying—but we touch, we talk, not through the Machine.”

He kissed her.

“We have come back to our own. We die, but we have recaptured life, as it was in Wessex, when Alfrid overthrew the Danes. We know what they know outside, they who dwelt in the cloud that is the colour of a pearl.”

“But Kuno, is it true? Are there still men on the surface of the earth? Is this—tunnel, this poisoned darkness—really not the end?”

He replied:
“I have seen them, spoken to them, loved them. They are hiding in the midst and the ferns until our civilization stops. Today they are the Homeless—tomorrow—”

“Oh, tomorrow—some fool will start the Machine again, tomorrow.”

“Never,” said Kuno, “never. Humanity has learnt its lesson.”

As he spoke, the whole city was broken like a honeycomb. An air-ship had sailed in through the vomitory into a ruined wharf. It crashed downwards, exploding as it went, rending gallery after gallery with its wings of steel. For a moment they saw the nations of the dead, and, before they joined them, scraps of the untainted sky.
As We May Think

—Vannevar Bush, 1945

This has not been a scientist’s war; it has been a war in which all have had a part. The scientists, burying their old professional competition in the demand of a common cause, have shared greatly and learned much. It has been exhilarating to work in effective partnership. Now, for many, this appears to be approaching an end. What are the scientists to do next?

For the biologists, and particularly for the medical scientists, there can be little indecision, for their war has hardly required them to leave the old paths. Many indeed have been able to carry on their war research in their familiar peacetime laboratories. Their objectives remain much the same.

It is the physicists who have been thrown most violently off stride, who have left academic pursuits for the making of strange destructive gadgets, who have had to devise new methods for their unanticipated assignments. They have done their part on the devices that made it possible to turn back the enemy, have worked in combined effort with the physicists of our allies. They have felt within themselves the stir of achievement. They have been part of a great team. Now, as peace approaches, one asks where they will find objectives worthy of their best.

1

Of what lasting benefit has been man’s use of science and of the new instruments which his research brought into existence? First, they have increased his control of his material environment. They have improved his food, his clothing, his shelter; they have increased his security and released him partly from the bondage of bare existence. They have given him increased knowledge of his own biological processes so that he has had a progressive freedom from disease and an increased span of life. They are illuminating

A scientist of the future records experiments with a tiny camera fitted with a universal focus lens. The small square in the eyeglass at the left lights the object.

This article appeared in the July 1945 edition of The Atlantic Monthly. The illustrations come from the Life magazine condensation of that article of September 10, 1945.
the interactions of his physiological and psychological functions, giving the promise of an improved mental health.

Science has provided the swiftest communication between individuals; it has provided a record of ideas and has enabled man to manipulate and to make extracts from that record so that knowledge evolves and endures throughout the life of a race rather than that of an individual.

There is a growing mountain of research. But there is increased evidence that we are being bogged down today as specialization extends. The investigator is staggered by the findings and conclusions of thousands of other workers—conclusions which he cannot find time to grasp, much less to remember, as they appear. Yet specialization becomes increasingly necessary for progress, and the effort to bridge between disciplines is correspondingly superficial.

Professionally our methods of transmitting and reviewing the results of research are generations old and by now are totally inadequate for their purpose. If the aggregate time spent in writing scholarly works and in reading them could be evaluated, the ratio between these amounts of time might well be startling. Those who conscientiously attempt to keep abreast of current thought, even in restricted fields, by close and continuous reading might well shy away from an examination calculated to show how much of the previous month’s efforts could be produced on call. Mendel’s concept of the laws of genetics was lost to the world for a generation because his publication did not reach the few who were capable of grasping and extending it; and this sort of catastrophe is undoubtedly being repeated all about us, as truly significant attainments become lost in the mass of the inconsequential.

The difficulty seems to be, not so much that we publish unduly in view of the extent and variety of present day interests, but rather that publication has been extended far beyond our present ability to make real use of the record. The summation of human experience is being expanded at a prodigious rate, and the means we use for threading through the consequent maze to the momentarily important item is the same as was used in the days of square-rigged ships.

But there are signs of a change as new and powerful instrumentalities come into use. Photocells capable of seeing things in a physical sense, advanced photography which can record what is seen or even what is not, thermionic tubes capable of controlling potent forces under the guidance of less power than a mosquito uses to vibrate his wings, cathode ray tubes rendering visible an occurrence so brief that by comparison a microsecond is a long time, relay combinations which will carry out involved sequences of movements more reliably than any human operator and thousands of times as fast—there are plenty of mechanical aids with which to effect a transformation in scientific records.

Two centuries ago Leibnitz invented a calculating machine which embodied most of the essential features of recent keyboard devices, but it could not then come into use. The economics of the situation were against it: the labor involved in constructing it, before the days of mass production, exceeded the labor to be saved by its use, since all it could accomplish could be

SUPERSECRETARY OF THE COMING AGE, the machine contemplated here would take dictation, type it automatically, and even talk back if the author wanted to review what he had just said. It is somewhat similar to the Voder seen at the New York World’s Fair. Like all machines suggested by the diagrams in this article, it is not yet in existence.
duplicated by sufficient use of pencil and paper. Moreover, it would have been subject to frequent breakdown, so that it could not have been depended upon; for at that time and long after, complexity and unreliability were synonymous.

Babbage, even with remarkably generous support for his time, could not produce his great arithmetical machine. His idea was sound enough, but construction and maintenance costs were then too heavy. Had a Pharaoh been given detailed and explicit designs of an automobile, and had he understood them completely, it would have taxed the resources of his kingdom to have fashioned the thousands of parts for a single car, and that car would have broken down on the first trip to Giza.

Machines with interchangeable parts can now be constructed with great economy of effort. In spite of much complexity, they perform reliably. Witness the humble typewriter, or the movie camera, or the automobile. Electrical contacts have ceased to stick when thoroughly understood. Note the automatic telephone exchange, which has hundreds of thousands of such contacts, and yet is reliable. A spider web of metal, sealed in a thin glass container, a wire heated to brilliant glow, in short, the thermionic tube of radio sets, is made by the hundred million, tossed about in packages, plugged into sockets—and it works! Its gossamer parts, the precise location and alignment involved in its construction, would have occupied a master craftsman of the guild for months; now it is built for thirty cents. The world has arrived at an age of cheap complex devices of great reliability; and something is bound to come of it.

A record if it is to be useful to science, must be continuously extended, it must be stored, and above all it must be consulted. Today we make the record conventionally by writing and photography, followed by printing; but we also record on film, on wax disks, and on magnetic wires. Even if utterly new recording procedures do not appear, these present ones are certainly in the process of modification and extension.

Certainly progress in photography is not going to stop. Faster material and lenses, more automatic cameras, finer-grained sensitive compounds to allow an extension of the minicamera idea, are all imminent. Let us project this trend ahead to a logical, if not inevitable, outcome. The camera hound of the future wears on his forehead a lump a little larger than a walnut. It takes pictures 3 millimeters square, later to be projected or enlarged, which after all involves only a factor of 10 beyond present practice. The lens is of universal focus, down to any distance accommodated by the unaided eye, simply because it is of short focal length. There is a built-in photocell on the walnut such as we now have on at least one camera, which automatically adjusts exposure for a wide range of illumination. There is film in the walnut for a hundred exposures, and the spring for operating its shutter and shifting its film is wound once for all when the film clip is inserted. It produces its result in full color. It may well be stereoscopic, and record with two spaced glass eyes, for striking improvements in stereoscopic technique are just around the corner.

The cord which trips its shutter may reach down a man’s sleeve within easy reach of his fingers. A quick squeeze, and the picture is taken. On a pair of ordinary glasses is a square of fine lines near the top of one lens, where it is out of the way of ordinary vision. When an object appears in that square, it is lined up for its picture. As the scientist of the future moves about the laboratory or the field, every time he looks at something worthy of the record, he trips the shutter and in it goes, without even an audible click. Is this all fantastic? The only fantastic thing about it is the idea of making as many pictures as would result from its use.

Will there be dry photography? It is already here in two forms. When Brady made his Civil War pictures, the plate had to be wet at the time of exposure. Now it has to be wet during development instead. In the future perhaps it need not be wetted at all. There have long been films impregnated with diazo dyes which form a picture without development, so that it is already there as soon as the camera has been operated. An exposure to ammonia gas destroys the unexposed dye, and the picture can then be taken out
into the light and examined. The process is now slow, but someone may speed it up, and it has no grain difficulties such as now keep photographic researchers busy. Often it would be advantageous to be able to snap the camera and to look at the picture immediately.

Another process now in use is also slow, and more or less clumsy. For fifty years impregnated papers have been used which turn dark at every point where an electrical contact touches them, by reason of the chemical change thus produced in an iodine compound included in the paper. They have been used to make records, for a pointer moving across them can leave a trail behind. If the electrical potential on the pointer is varied as it moves, the line becomes light or dark in accordance with the potential.

This scheme is now used in facsimile transmission. The pointer draws a set of closely spaced lines across the paper one after another. As it moves, its potential is varied in accordance with a varying current received over wires from a distant station, where these variations are produced by a photocell which is similarly scanning a picture. At every instant the darkness of the line being drawn is made equal to the darkness of the point on the picture being observed by the photocell. Thus, when the whole picture has been covered, a replica appears at the receiving end.

A scene itself can be just as well looked over line by line by the photocell in this way as can a photograph of the scene. This whole apparatus constitutes a camera, with the added feature, which can be dispensed with if desired, of making its picture at a distance. It is slow, and the picture is poor in detail. Still, it does give another process of dry photography, in which the picture is finished as soon as it is taken.

It would be a brave man who would predict that such a process will always remain clumsy, slow, and faulty in detail. Television equipment today transmits sixteen reasonably good pictures a second, and it involves only two essential differences from the process described above. For one, the record is made by a moving beam of electrons rather than a moving pointer, for the reason that an electron beam can sweep across the picture very rapidly indeed. The other difference involves merely the use of a screen which glows momentarily when the electrons hit, rather than a chemically treated paper or film which is permanently altered. This speed is necessary in television, for motion pictures rather than stills are the object.

Use chemically treated film in place of the glowing screen, allow the apparatus to transmit one picture only rather than a succession, and a rapid camera for dry photography results. The treated film needs to be far faster in action than present examples, but it probably could be. More serious is the objection that this scheme would involve putting the film inside a vacuum chamber, for electron beams behave normally only in such a rarefied environment. This difficulty could be avoided by allowing the electron beam to play on one side of a partition, and by pressing the film against the other side, if this partition were such as to allow the electrons to go through perpendicular to its surface, and to prevent them from spreading out sideways. Such partitions, in crude form, could certainly be constructed, and they will hardly hold up the general development.

Like dry photography, microphotography still has a long way to go. The basic scheme of reducing the size of the record, and examining it by projection rather than directly, has possibilities too great to be ignored. The combination of optical projection and photographic reduction is already producing some results in microfilm for scholarly purposes, and the potentialities are highly suggestive. Today, with microfilm, reductions by a linear factor of 20 can be employed and still produce full clarity when the material is re-enlarged for examination. The limits are set by the graininess of the film, the excellence of the optical system, and the efficiency of the light sources employed. All of these are rapidly improving.

Assume a linear ratio of 100 for future use. Consider film of the same thickness as paper, although thinner film will certainly be usable. Even under these conditions there would be a total factor of 10,000
between the bulk of the ordinary record on books, and its microfilm replica. The *Encyclopædia Britannica* could be reduced to the volume of a matchbox. A library of a million volumes could be compressed into one end of a desk. If the human race has produced since the invention of movable type a total record, in the form of magazines, newspapers, books, tracts, advertising blurbs, correspondence, having a volume corresponding to a billion books, the whole affair, assembled and compressed, could be lugged off in a moving van. Mere compression, of course, is not enough; one needs not only to make and store a record but also be able to consult it, and this aspect of the matter comes later. Even the modern great library is not generally consulted; it is nibbled at by a few.

Compression is important, however, when it comes to costs. The material for the microfilm *Britannica* would cost a nickel, and it could be mailed anywhere for a cent. What would it cost to print a million copies? To print a sheet of newspaper, in a large edition, costs a small fraction of a cent. The entire material of the *Britannica* in reduced microfilm form would go on a sheet eight and one-half by eleven inches. Once it is available, with the photographic reproduction methods of the future, duplicates in large quantities could probably be turned out for a cent apiece beyond the cost of materials. The preparation of the original copy? That introduces the next aspect of the subject.

To make the record, we now push a pencil or tap a typewriter. Then comes the process of digestion and correction, followed by an intricate process of typesetting, printing, and distribution. To consider the first stage of the procedure, will the author of the future cease writing by hand or typewriter and talk directly to the record? He does so indirectly, by talking to a stenographer or a wax cylinder; but the elements are all present if he wishes to have his talk directly produce a typed record. All he needs to do is to take advantage of existing mechanisms and to alter his language.

At a recent World Fair a machine called a Voder was shown. A girl stroked its keys and it emitted recognizable speech. No human vocal chords entered into the procedure at any point; the keys simply combined some electrically produced vibrations and passed these on to a loud-speaker. In the Bell Laboratories there is the converse of this machine, called a Vocoder. The loudspeaker is replaced by a microphone, which picks up sound. Speak to it, and the corresponding keys move. This may be one element of the postulated system.

The other element is found in the stenotype, that somewhat disconcerting device encountered usually at public meetings. A girl strokes its keys languidly and looks about the room and sometimes at the speaker with a disquieting gaze. From it emerges a typed strip which records in a phonetically simplified language a record of what the speaker is supposed to have said. Later this strip is retyped into ordinary language, for in its nascent form it is intelligible only to the initiated. Combine these two elements, let the Vocoder run the stenotype, and the result is a machine which types when talked to.

Our present languages are not especially adapted to this sort of mechanization, it is true. It is strange that the inventors of universal languages have not seized upon the idea of producing one which better fitted the technique for transmitting and recording speech. Mechanization may yet force the issue, especially in the scientific field; whereupon scientific jargon would become still less intelligible to the layman.

One can now picture a future investigator in his laboratory. His hands are free, and he is not anchored. As he moves about and observes, he photographs and comments. Time is automatically recorded to tie the two records together. If he goes into the field, he may be connected by radio to his recorder. As he ponders over his notes in the evening, he again talks his comments into the record. His typed record, as well as his photographs, may both be in miniature, so that he projects them for examination.
Much needs to occur, however, between the collection of data and observations, the extraction of parallel material from the existing record, and the final insertion of new material into the general body of the common record. For mature thought there is no mechanical substitute. But creative thought and essentially repetitive thought are very different things. For the latter there are, and may be, powerful mechanical aids.

Adding a column of figures is a repetitive thought process, and it was long ago properly relegated to the machine. True, the machine is sometimes controlled by a keyboard, and thought of a sort enters in reading the figures and poking the corresponding keys, but even this is avoidable. Machines have been made which will read typed figures by photocells and then depress the corresponding keys; these are combinations of photocells for scanning the type, electric circuits for sorting the consequent variations, and relay circuits for interpreting the result into the action of solenoids to pull the keys down.

All this complication is needed because of the clumsy way in which we have learned to write figures. If we recorded them positionally, simply by the configuration of a set of dots on a card, the automatic reading mechanism would become comparatively simple. In fact if the dots are holes, we have the punched-card machine long ago produced by Hollerith for the purposes of the census, and now used throughout business. Some types of complex businesses could hardly operate without these machines.

Adding is only one operation. To perform arithmetical computation involves also subtraction, multiplication, and division, and in addition some method for temporary storage of results, removal from storage for further manipulation, and recording of final results by printing. Machines for these purposes are now of two types: keyboard machines for accounting and the like, manually controlled for the insertion of data, and usually automatically controlled as far as the sequence of operations is concerned; and punched-card machines in which separate operations are usually delegated to a series of machines, and the cards then transferred bodily from one to another. Both forms are very useful; but as far as complex computations are concerned, both are still in embryo.

Rapid electrical counting appeared soon after the physicists found it desirable to count cosmic rays. For their own purposes the physicists promptly constructed thermionic-tube equipment capable of counting electrical impulses at the rate of 100,000 a second. The advanced arithmetical machines of the future will be electrical in nature, and they will perform at 100 times present speeds, or more.

Moreover, they will be far more versatile than present commercial machines, so that they may readily be adapted for a wide variety of operations. They will be controlled by a control card or film, they will select their own data and manipulate it in accordance with the instructions thus inserted, they will perform complex arithmetical computations at exceedingly high speeds, and they will record results in such form as to be readily available for distribution or for later further manipulation. Such machines will have enormous appetites. One of them will take instructions and data from a whole roomful of girls armed with simple keyboard punches, and will deliver sheets of computed results every few minutes. There will always be plenty of things to compute in the detailed affairs of millions of people doing complicated things.

The repetitive processes of thought are not confined however, to matters of arithmetic and statistics. In fact, every time one combines and records facts in accordance with established logical processes, the creative aspect of thinking is concerned only with the selection of the data and the process to be employed and the manipulation thereafter is repetitive in nature and hence a fit matter to be relegated to the machine. Not so much has been done along these lines, beyond the bounds of arithmetic, as might be done, primarily because of the economics of the situation. The needs of business and the extensive market obviously waiting, assured the advent of mass-produced arithmetical machines just as soon as production methods were sufficiently advanced.
With machines for advanced analysis no such situation existed; for there was and is no extensive market; the users of advanced methods of manipulating data are a very small part of the population. There are, however, machines for solving differential equations—and functional and integral equations, for that matter. There are many special machines, such as the harmonic synthesizer which predicts the tides. There will be many more, appearing certainly first in the hands of the scientist and in small numbers.

If scientific reasoning were limited to the logical processes of arithmetic, we should not get far in our understanding of the physical world. One might as well attempt to grasp the game of poker entirely by the use of the mathematics of probability. The abacus, with its beads strung on parallel wires, led the Arabs to positional numeration and the concept of zero many centuries before the rest of the world; and it was a useful tool—so useful that it still exists.

It is a far cry from the abacus to the modern keyboard accounting machine. It will be an equal step to the arithmetical machine of the future. But even this new machine will not take the scientist where he needs to go. Relief must be secured from laborious detailed manipulation of higher mathematics as well, if the users of it are to free their brains for something more than repetitive detailed transformations in accordance with established rules. A mathematician is not a man who can readily manipulate figures; often he cannot. He is not even a man who can readily perform the transformations of equations by the use of calculus. He is primarily an individual who is skilled in the use of symbolic logic on a high plane, and especially he is a man of intuitive judgment in the choice of the manipulative processes he employs.

All else he should be able to turn over to his mechanism, just as confidently as he turns over the propelling of his car to the intricate mechanism under the hood. Only then will mathematics be practically effective in bringing the growing knowledge of atomistics to the useful solution of the advanced problems of chemistry, metallurgy, and biology. For this reason there still come more machines to handle advanced mathematics for the scientist. Some of them will be sufficiently bizarre to suit the most fastidious connoisseur of the present artifacts of civilization.

The scientist, however, is not the only person who manipulates data and examines the world about him by the use of logical processes, although he sometimes preserves this appearance by adopting into the fold anyone who becomes logical, much in the manner in which a British labor leader is elevated to knighthood. Whenever logical processes of thought are employed—that is, whenever thought for a time runs along an accepted groove—there is an opportunity for the machine. Formal logic used to be a keen instrument in the hands of the teacher in his trying of students’ souls. It is readily possible to construct a machine which will manipulate premises in accordance with formal logic, simply by the clever use of relay circuits. Put a set of premises into such a device and turn the crank, and it will readily pass out conclusion after conclusion, all in accordance with logical law, and with no more slips than would be expected of a keyboard adding machine.

Logic can become enormously difficult, and it would undoubtedly be well to produce more assurance in its use. The machines for higher analysis have usually been equation solvers. Ideas are beginning to appear for equation transformers, which will rearrange the relationship expressed by an equation in accordance with strict and rather advanced logic. Progress is inhibited by the exceedingly crude way in which mathematicians express their relationships. They employ a symbolism which grew like Topsy and has little consistency; a strange fact in that most logical field.

A new symbolism, probably positional, must apparently precede the reduction of mathematical transformations to machine processes. Then, on beyond the strict logic of the mathematician, lies the application of logic in everyday affairs. We may some day click off arguments on a machine with the same
assurance that we now enter sales on a cash register. But the machine of logic will not look like a cash register, even of the streamlined model.

So much for the manipulation of ideas and their insertion into the record. Thus far we seem to be worse off than before—for we can enormously extend the record; yet even in its present bulk we can hardly consult it. This is a much larger matter than merely the extraction of data for the purposes of scientific research; it involves the entire process by which man profits by his inheritance of acquired knowledge. The prime action of use is selection, and here we are halting indeed. There may be millions of fine thoughts, and the account of the experience on which they are based, all encased within stone walls of acceptable architectural form; but if the scholar can get at only one a week by diligent search, his syntheses are not likely to keep up with the current scene.

Selection, in this broad sense, is a stone adze in the hands of a cabinetmaker. Yet, in a narrow sense and in other areas, something has already been done mechanically on selection. The personnel officer of a factory drops a stack of a few thousand employee cards into a selecting machine, sets a code in accordance with an established convention, and produces in a short time a list of all employees who live in Trenton and know Spanish. Even such devices are much too slow when it comes, for example, to matching a set of fingerprints with one of five million on file. Selection devices of this sort will soon be speeded up from their present rate of reviewing data at a few hundred a minute. By the use of photocells and microfilm they will survey items at the rate of a thousand a second, and will print out duplicates of those selected.

This process, however, is simple selection: it proceeds by examining in turn every one of a large set of items, and by picking out those which have certain specified characteristics. There is another form of selection best illustrated by the automatic telephone exchange. You dial a number and the machine selects and connects just one of a million possible stations. It does not run over them all. It pays attention only to a class given by a first digit, then only to a subclass of this given by the second digit, and so on; and thus proceeds rapidly and almost unerringly to the selected station. It requires a few seconds to make the selection, although the process could be speeded up if increased speed were economically warranted. If necessary, it could be made extremely fast by substituting thermionic-tube switching for mechanical switching, so that the full selection could be made in one one-hundredth of a second. No one would wish to spend the money necessary to make this change in the telephone system, but the general idea is applicable elsewhere.

Take the prosaic problem of the great department store. Every time a charge sale is made, there are a number of things to be done. The inventory needs to be revised, the salesman needs to be given credit for the sale, the general accounts need an entry, and, most important, the customer needs to be charged. A central records device has been developed in which much of this work is done conveniently. The salesman places on a stand the customer’s identification card, his own card, and the card taken from the article sold—all punched cards. When he pulls a lever, contacts are made through the holes, machinery at a central point makes the necessary computations and entries, and the proper receipt is printed for the salesman to pass to the customer.

But there may be ten thousand charge customers doing business with the store, and before the full operation can be completed someone has to select the right card and insert it at the central office. Now rapid selection can slide just the proper card into position in an instant or two, and return it afterward. Another difficulty occurs, however. Someone must read a total on the card, so that the machine can add its computed item to it. Conceivably the cards might be of the dry photography type I have described. Existing totals could then be read by photocell, and the new total entered by an electron beam.

The cards may be in miniature, so that they occupy little space. They must move quickly. They need not be transferred far, but merely into position so that the photocell and recorder can operate on them.
Positional dots can enter the data. At the end of the month a machine can readily be made to read these and to print an ordinary bill. With tube selection, in which no mechanical parts are involved in the switches, little time need be occupied in bringing the correct card into use—a second should suffice for the entire operation. The whole record on the card may be made by magnetic dots on a steel sheet if desired, instead of dots to be observed optically, following the scheme by which Poulsen long ago put speech on a magnetic wire. This method has the advantage of simplicity and ease of erasure. By using photography, however one can arrange to project the record in enlarged form and at a distance by using the process common in television equipment.

One can consider rapid selection of this form, and distant projection for other purposes. To be able to key one sheet of a million before an operator in a second or two, with the possibility of then adding notes thereto, is suggestive in many ways. It might even be of use in libraries, but that is another story. At any rate, there are now some interesting combinations possible. One might, for example, speak to a microphone, in the manner described in connection with the speech controlled typewriter, and thus make his selections. It would certainly beat the usual file clerk.

6

The real heart of the matter of selection, however, goes deeper than a lag in the adoption of mechanisms by libraries, or a lack of development of devices for their use. Our ineptitude in getting at the record is largely caused by the artificiality of systems of indexing. When data of any sort are placed in storage, they are filed alphabetically or numerically, and information is found (when it is) by tracing it down from subclass to subclass. It can be in only one place, unless duplicates are used; one has to have rules as to which path will locate it, and the rules are cumbersome. Having found one item, moreover, one has to emerge from the system and re-enter on a new path.

The human mind does not work that way. It operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain. It has other characteristics, of course; trails that are not frequently followed are prone to fade, items are not fully permanent, memory is transitory. Yet the speed of action, the intricacy of trails, the detail of mental pictures, is awe-inspiring beyond all else in nature.

Man cannot hope fully to duplicate this mental process artificially, but he certainly ought to be able to learn from it. In minor ways he may even improve, for his records have relative permanency. The first idea, however, to be drawn from the analogy concerns selection. Selection by association, rather than indexing, may yet be mechanized. One cannot hope thus to equal the speed and flexibility with which the mind follows an associative trail, but it should be possible to beat the mind decisively in regard to the permanence and clarity of the items resurrected from storage.

Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, “memex” will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory.

It consists of a desk, and while it can presumably be operated from a distance, it is primarily the piece of furniture at which he works. On the top are slanting translucent screens, on which material can be projected for convenient reading. There is a keyboard, and sets of buttons and levers. Otherwise it looks like an ordinary desk.

In one end is the stored material. The matter of bulk is well taken care of by improved microfilm. Only a small part of the interior of the memex is devoted to storage, the rest to mechanism. Yet if the user inserted
5000 pages of material a day it would take him hundreds of years to fill the repository, so he can be profligate and enter material freely.

Most of the memex contents are purchased on microfilm ready for insertion. Books of all sorts, pictures, current periodicals, newspapers, are thus obtained and dropped into place. Business correspondence takes the same path. And there is provision for direct entry. On the top of the memex is a transparent platen. On this are placed longhand notes, photographs, memoranda, all sorts of things. When one is in place, the depression of a lever causes it to be photographed onto the next blank space in a section of the memex film, dry photography being employed.

There is, of course, provision for consultation of the record by the usual scheme of indexing. If the user wishes to consult a certain book, he taps its code on the keyboard, and the title page of the book promptly appears before him, projected onto one of his viewing positions. Frequently-used codes are mnemonic, so that he seldom consults his code book; but when he does, a single tap of a key projects it for his use. Moreover, he has supplemental levers. On deflecting one of these levers to the right he runs through the book before him, each page in turn being projected at a speed which just allows a recognizing glance at each. If he deflects it further to the right, he steps through the book 10 pages at a time; still further at 100 pages at a time. Deflection to the left gives him the same control backwards.

A special button transfers him immediately to the first page of the index. Any given book of his library can thus be called up and consulted with far greater facility than if it were taken from a shelf. As he has several projection positions, he can leave one item in position while he calls up another. He can add marginal notes and comments, taking advantage of one possible type of dry photography, and it could even be arranged so that he can do this by a stylus scheme, such as is now employed in the telautograph seen in railroad waiting rooms, just as though he had the physical page before him.

All this is conventional, except for the projection forward of present-day mechanisms and gadgetry. It affords an immediate step, however, to associative indexing, the basic idea of which is a provision whereby any item may be caused at will to select immediately and automatically another. This is the essential feature of the memex. The process of tying two items together is the important thing.

When the user is building a trail, he names it, inserts the name in his code book, and taps it out on his keyboard. Before him are the two items to be joined, projected onto adjacent viewing positions. At the bottom of each there are a number of blank code spaces, and a pointer is set to indicate one of these on each item. The user taps a single key, and the items are permanently joined. In each code space appears the code word. Out of view, but also in the code space, is inserted a set of dots for photocell viewing; and on each item these dots by their positions designate the index number of the other item.

Thereafter, at any time, when one of these items is in view, the other can be instantly recalled merely by tapping a button below the corresponding code space. Moreover, when numerous items have been thus joined together to form a trail, they can be reviewed in turn, rapidly
or slowly, by deflecting a lever like that used for turning the pages of a book. It is exactly as though the physical items had been gathered together from widely separated sources and bound together to form a new book. It is more than this, for any item can be joined into numerous trails.

The owner of the memex, let us say, is interested in the origin and properties of the bow and arrow. Specifically he is studying why the short Turkish bow was apparently superior to the English long bow in the skirmishes of the Crusades. He has dozens of possibly pertinent books and articles in his memex. First he runs through an encyclopedia, finds an interesting but sketchy article, leaves it projected. Next, in a history, he finds another pertinent item, and ties the two together. Thus he goes, building a trail of many items. Occasionally he inserts a comment of his own, either linking it into the main trail or joining it by a side trail to a particular item. When it becomes evident that the elastic properties of available materials had a great deal to do with the bow, he branches off on a side trail which takes him through textbooks on elasticity and tables of physical constants. He inserts a page of longhand analysis of his own. Thus he builds a trail of his interest through the maze of materials available to him.

And his trails do not fade. Several years later, his talk with a friend turns to the queer ways in which a people resist innovations, even of vital interest. He has an example, in the fact that the outraged Europeans still failed to adopt the Turkish bow. In fact he has a trail on it. A touch brings up the code book. Tapping a few keys projects the head of the trail. A lever runs through it at will, stopping at interesting items, going off on side excursions. It is an interesting trail, pertinent to the discussion. So he sets a reproducer in action, photographs the whole trail out, and passes it to his friend for insertion in his own memex, there to be linked into the more general trail.

Wholly new forms of encyclopedias will appear, ready made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified. The lawyer has at his touch the associated opinions and decisions of his whole experience, and of the experience of friends and authorities. The patent attorney has on call the millions of issued patents, with familiar trails to every point of his client’s interest. The physician, puzzled by a patient’s reactions, strikes the trail established in studying an earlier similar case, and runs rapidly through analogous case histories, with side references to the classics for the pertinent anatomy and histology. The chemist, struggling with the synthesis of an organic compound, has all the chemical literature before him in his laboratory, with trails following the analogies of compounds, and side trails to their physical and chemical behavior.

The historian, with a vast chronological account of a people, parallels it with a skip trail which stops only on the salient items, and can follow at any time contemporary trails which lead him all over civilization at a particular epoch. There is a new profession of trail blazers, those who find delight in the task of establishing useful trails through the enormous mass of the common record. The inheritance from the master becomes, not only his additions to the world’s record, but for his disciples the entire scaffolding by which they were erected.
Thus science may implement the ways in which man produces, stores, and consults the record of the race. It might be striking to outline the instrumentalities of the future more spectacularly, rather than to stick closely to methods and elements now known and undergoing rapid development, as has been done here. Technical difficulties of all sorts have been ignored, certainly, but also ignored are means as yet unknown which may come any day to accelerate technical progress as violently as did the advent of the thermionic tube. In order that the picture may not be too commonplace, by reason of sticking to present-day patterns, it may be well to mention one such possibility, not to prophesy but merely to suggest, for prophecy based on extension of the known has substance, while prophecy founded on the unknown is only a doubly involved guess.

All our steps in creating or absorbing material of the record proceed through one of the senses—the tactile when we touch keys, the oral when we speak or listen, the visual when we read. Is it not possible that some day the path may be established more directly?

We know that when the eye sees, all the consequent information is transmitted to the brain by means of electrical vibrations in the channel of the optic nerve. This is an exact analogy with the electrical vibrations which occur in the cable of a television set: they convey the picture from the photocells which see it to the radio transmitter from which it is broadcast. We know further that if we can approach that cable with the proper instruments, we do not need to touch it; we can pick up those vibrations by electrical induction and thus discover and reproduce the scene which is being transmitted, just as a telephone wire may be tapped for its message.

The impulses which flow in the arm nerves of a typist convey to her fingers the translated information which reaches her eye or ear, in order that the fingers may be caused to strike the proper keys. Might not these currents be intercepted, either in the original form in which information is conveyed to the brain, or in the marvelously metamorphosed form in which they then proceed to the hand?

By bone conduction we already introduce sounds: into the nerve channels of the deaf in order that they may hear. Is it not possible that we may learn to introduce them without the present cumbersomeness of first transforming electrical vibrations to mechanical ones, which the human mechanism promptly transforms back to the electrical form? With a couple of electrodes on the skull the encephalograph now produces pen-and-ink traces which bear some relation to the electrical phenomena going on in the brain itself. True, the record is unintelligible, except as it points out certain gross misfunctioning of the cerebral mechanism; but who would now place bounds on where such a thing may lead?

In the outside world, all forms of intelligence whether of sound or sight, have been reduced to the form of varying currents in an electric circuit in order that they may be transmitted. Inside the human frame exactly the same sort of process occurs. Must we always transform to mechanical movements in order to proceed from one electrical phenomenon to another? It is a suggestive thought, but it hardly warrants prediction without losing touch with reality and immediateness.

Presumably man’s spirit should be elevated if he can better review his shady past and analyze more completely and objectively his present problems. He has built a civilization so complex that he needs to mechanize his records more fully if he is to push his experiment to its logical conclusion and not merely become bogged down part way there by overtaxing his limited memory. His excursions may be more enjoyable if he can reacquire the privilege of forgetting the manifold things he does not need to have immediately at hand, with some assurance that he can find them again if they prove important.

The applications of science have built man a well-supplied house, and are teaching him to live healthily therein. They have enabled him to throw masses of people against one another with cruel weapons. They may yet allow him truly to encompass the great record and to grow in the wisdom of race experience. He
may perish in conflict before he learns to wield that record for his true good. Yet, in the application of science to the needs and desires of man, it would seem to be a singularly unfortunate stage at which to terminate the process, or to lose hope as to the outcome.

As Director of the Office of Scientific Research and Development, Dr. Vannevar Bush has coordinated the activities of some six thousand leading American scientists in the application of science to warfare. In this significant article he holds up an incentive for scientists when the fighting has ceased. He urges that men of science should then turn to the massive task of making more accessible our bewildering store of knowledge. For years inventions have extended man’s physical powers rather than the powers of his mind. Trip hammers that multiply the fists, microscopes that sharpen the eye, and engines of destruction and detection are new results, but not the end results, of modern science. Now, says Dr. Bush, instruments are at hand which, if properly developed, will give man access to and command over the inherited knowledge of the ages. The perfection of these pacific instruments should be the first objective of our scientists as they emerge from their war work. Like Emerson’s famous address of 1837 on “The American Scholar,” this paper by Dr. Bush calls for a new relationship between thinking man and the sum of our knowledge.

—THE EDITOR
1. The Imitation Game

I propose to consider the question, “Can machines think?” This should begin with definitions of the meaning of the terms “machine” and “think.” The definitions might be framed so as to reflect so far as possible the normal use of the words, but this attitude is dangerous. If the meaning of the words “machine” and “think” are to be found by examining how they are commonly used it is difficult to escape the conclusion that the meaning and the answer to the question, “Can machines think?” is to be sought in a statistical survey such as a Gallup poll. But this is absurd. Instead of attempting such a definition I shall replace the question by another, which is closely related to it and is expressed in relatively unambiguous words.

The new form of the problem can be described in terms of a game which we call the “imitation game.” It is played with three people, a man (A), a woman (B), and an interrogator (C) who may be of either sex. The interrogator stays in a room apart from the other two. The object of the game for the interrogator is to determine which of the other two is the man and which is the woman. He knows them by labels X and Y, and at the end of the game he says either “X is A and Y is B” or “X is B and Y is A.” The interrogator is allowed to put questions to A and B thus:

C: Will X please tell me the length of his or her hair?

Now suppose X is actually A, then A must answer. It is A’s object in the game to try and cause C to make the wrong identification. His answer might therefore be:

“My hair is shingled, and the longest strands are about nine inches long.”

In order that tones of voice may not help the interrogator the answers should be written, or better still, typewritten. The ideal arrangement is to have a teleprinter communicating between the two rooms. Alternatively the question and answers can be repeated by an intermediary. The object of the game for the third player (B) is to help the interrogator. The best strategy for her is probably to give truthful answers. She can add such things as “I am the woman, don’t listen to him!” to her answers, but it will avail nothing as the man can make similar remarks.

We now ask the question, “What will happen when a machine takes the part of A in this game?” Will the interrogator decide wrongly as often when the game is played like this as he does when the game is played between a man and a woman? These questions replace our original, “Can machines think?”

2. Critique of the New Problem

As well as asking, “What is the answer to this new form of the question,” one may ask, “Is this new question a worthy one to investigate?” This latter question we investigate without further ado, thereby cutting short an infinite regress.

The new problem has the advantage of drawing a fairly sharp line between the physical and the intellectual capacities of a man. No engineer or chemist claims to be able to produce a material which is indistinguishable from the human skin. It is possible that at some time this might be done, but even supposing this invention available we should feel there was little point in trying to make a “thinking machine” more human by dressing it up in such artificial flesh. The form in which we have set the problem reflects this fact in the condition which prevents the interrogator from seeing or touching the other
competitors, or hearing their voices. Some other advantages of the proposed criterion may be shown up by specimen questions and answers. Thus:

Q: Please write me a sonnet on the subject of the Forth Bridge.
A: Count me out on this one. I never could write poetry.

Q: Add 34957 to 70764.
A: (Pause about 30 seconds and then give as answer) 105621.

Q: Do you play chess?
A: Yes.

Q: I have K at my K1, and no other pieces. You have only K at K6 and R at R1. It is your move. What do you play?
A: (After a pause of 15 seconds) R-R8 mate.

The question and answer method seems to be suitable for introducing almost any one of the fields of human endeavour that we wish to include. We do not wish to penalise the machine for its inability to shine in beauty competitions, nor to penalise a man for losing in a race against an aeroplane. The conditions of our game make these disabilities irrelevant. The “witnesses” can brag, if they consider it advisable, as much as they please about their charms, strength or heroism, but the interrogator cannot demand practical demonstrations.

The game may perhaps be criticised on the ground that the odds are weighted too heavily against the machine. If the man were to try and pretend to be the machine he would clearly make a very poor showing. He would be given away at once by slowness and inaccuracy in arithmetic. May not machines carry out something which ought to be described as thinking but which is very different from what a man does? This objection is a very strong one, but at least we can say that if, nevertheless, a machine can be constructed to play the imitation game satisfactorily, we need not be troubled by this objection.

It might be urged that when playing the “imitation game” the best strategy for the machine may possibly be something other than imitation of the behaviour of a man. This may be, but I think it is unlikely that there is any great effect of this kind. In any case there is no intention to investigate here the theory of the game, and it will be assumed that the best strategy is to try to provide answers that would naturally be given by a man.

3. The Machines concerned in the Game

The question which we put in §1 will not be quite definite until we have specified what we mean by the word “machine.” It is natural that we should wish to permit every kind of engineering technique to be used in our machines. We also wish to allow the possibility than an engineer or team of engineers may construct a machine which works, but whose manner of operation cannot be satisfactorily described by its constructors because they have applied a method which is largely experimental. Finally, we wish to exclude from the machines men born in the usual manner. It is difficult to frame the definitions so as to satisfy these three conditions. One might for instance insist that the team of engineers should be all of one sex, but this would not really be satisfactory, for it is probably possible to rear a complete individual from a single cell of the skin (say) of a man. To do so would be a feat of biological technique deserving of the very highest praise, but we would not be inclined to regard it as a case of “constructing a thinking machine.” This prompts us to abandon the requirement that every kind of technique should be permitted. We are the more ready to do so in view of the fact that the present interest in “thinking machines” has been aroused by a particular kind of machine, usually called an “electronic computer” or “digital computer.” Following this suggestion we only permit digital computers to take part in our game.
This restriction appears at first sight to be a very drastic one. I shall attempt to show that it is not so in reality. To do this necessitates a short account of the nature and properties of these computers.

It may also be said that this identification of machines with digital computers, like our criterion for “thinking,” will only be unsatisfactory if (contrary to my belief), it turns out that digital computers are unable to give a good showing in the game.

There are already a number of digital computers in working order, and it may be asked, “Why not try the experiment straight away? It would be easy to satisfy the conditions of the game. A number of interrogators could be used, and statistics compiled to show how often the right identification was given.” The short answer is that we are not asking whether all digital computers would do well in the game nor whether the computers at present available would do well, but whether there are imaginable computers which would do well. But this is only the short answer. We shall see this question in a different light later.

4. Digital Computers

The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations. He may also do his multiplications and additions on a “desk machine,” but this is not important.

If we use the above explanation as a definition we shall be in danger of circularity of argument. We avoid this by giving an outline of the means by which the desired effect is achieved. A digital computer can usually be regarded as consisting of three parts:

(i) Store.
(ii) Executive unit.
(iii) Control.

The store is a store of information, and corresponds to the human computer’s paper, whether this is the paper on which he does his calculations or that on which his book of rules is printed. In so far as the human computer does calculations in his bead a part of the store will correspond to his memory.

The executive unit is the part which carries out the various individual operations involved in a calculation. What these individual operations are will vary from machine to machine. Usually fairly lengthy operations can be done such as “Multiply 3540675445 by 7076345687” but in some machines only very simple ones such as “Write down 0” are possible.

We have mentioned that the “book of rules” supplied to the computer is replaced in the machine by a part of the store. It is then called the “table of instructions.” It is the duty of the control to see that these instructions are obeyed correctly and in the right order. The control is so constructed that this necessarily happens.

The information in the store is usually broken up into packets of moderately small size. In one machine, for instance, a packet might consist of ten decimal digits. Numbers are assigned to the parts of the store in which the various packets of information are stored, in some systematic manner. A typical instruction might say—

“Add the number stored in position 6809 to that in 4302 and put the result back into the latter storage position.”
Needless to say it would not occur in the machine expressed in English. It would more likely be coded in a form such as 6809430217. Here 17 says which of various possible operations is to be performed on the two numbers. In this case the operation is that described above, viz., “Add the number. . . .” It will be noticed that the instruction takes up 10 digits and so forms one packet of information, very conveniently. The control will normally take the instructions to be obeyed in the order of the positions in which they are stored, but occasionally an instruction such as

“Now obey the instruction stored in position 5606, and continue from there”

may be encountered, or again

“If position 4505 contains 0 obey next the instruction stored in 6707, otherwise continue straight on.”

Instructions of these latter types are very important because they make it possible for a sequence of operations to be replaced over and over again until some condition is fulfilled, but in doing so to obey, not fresh instructions on each repetition, but the same ones over and over again. To take a domestic analogy. Suppose Mother wants Tommy to call at the cobbler’s every morning on his way to school to see if her shoes are done, she can ask him afresh every morning. Alternatively she can stick up a notice once and for all in the hall which he will see when he leaves for school and which tells him to call for the shoes, and also to destroy the notice when he comes back if he has the shoes with him.

The reader must accept it as a fact that digital computers can be constructed, and indeed have been constructed, according to the principles we have described, and that they can in fact mimic the actions of a human computer very closely.

The book of rules which we have described our human computer as using is of course a convenient fiction. Actual human computers really remember what they have got to do. If one wants to make a machine mimic the behaviour of the human computer in some complex operation one has to ask him how it is done, and then translate the answer into the form of an instruction table. Constructing instruction tables is usually described as “programming.” To “programme a machine to carry out the operation A” means to put the appropriate instruction table into the machine so that it will do A.

An interesting variant on the idea of a digital computer is a “digital computer with a random element.” These have instructions involving the throwing of a die or some equivalent electronic process; one such instruction might for instance be, “Throw the die and put the resulting number into store 1000.” Sometimes such a machine is described as having free will (though I would not use this phrase myself), It is not normally possible to determine from observing a machine whether it has a random element, for a similar effect can be produced by such devices as making the choices depend on the digits of the decimal for .

Most actual digital computers have only a finite store. There is no theoretical difficulty in the idea of a computer with an unlimited store. Of course only a finite part can have been used at any one time. Likewise only a finite amount can have been constructed, but we can imagine more and more being added as required. Such computers have special theoretical interest and will be called infinitive capacity computers.

The idea of a digital computer is an old one. Charles Babbage, Lucasian Professor of Mathematics at Cambridge from 1828 to 1839, planned such a machine, called the Analytical Engine, but it was never completed. Although Babbage had all the essential ideas, his machine was not at that time such a very attractive prospect. The speed which would have been available would be definitely faster than a human computer but something like 100 times slower than the Manchester machine, itself one of the slower of the modern machines. The storage was to be purely mechanical, using wheels and cards.
The fact that Babbage’s Analytical Engine was to be entirely mechanical will help us to rid ourselves of a superstition. Importance is often attached to the fact that modern digital computers are electrical, and that the nervous system also is electrical. Since Babbage’s machine was not electrical, and since all digital computers are in a sense equivalent, we see that this use of electricity cannot be of theoretical importance. Of course electricity usually comes in where fast signalling is concerned, so that it is not surprising that we find it in both these connections. In the nervous system chemical phenomena are at least as important as electrical. In certain computers the storage system is mainly acoustic. The feature of using electricity is thus seen to be only a very superficial similarity. If we wish to find such similarities we should look rather for mathematical analogies of function.

5. Universality of Digital Computers

The digital computers considered in the last section may be classified amongst the “discrete-state machines.” These are the machines which move by sudden jumps or clicks from one quite definite state to another. These states are sufficiently different for the possibility of confusion between them to be ignored. Strictly speaking there, are no such machines. Everything really moves continuously. But there are many kinds of machine which can profitably be thought of as being discrete-state machines. For instance in considering the switches for a lighting system it is a convenient fiction that each switch must be definitely on or definitely off. There must be intermediate positions, but for most purposes we can forget about them. As an example of a discrete-state machine we might consider a wheel which clicks round through 120 once a second, but may be stopped by a lever which can be operated from outside; in addition a lamp is to light in one of the positions of the wheel. This machine could be described abstractly as follows. The internal state of the machine (which is described by the position of the wheel) may be $q_1$, $q_2$ or $q_3$. There is an input signal $i_0$ or $i_1$ (position of lever). The internal state at any moment is determined by the last state and input signal according to the table

<table>
<thead>
<tr>
<th>Input</th>
<th>Last State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$</td>
<td>$q_1$ $q_2$ $q_3$</td>
</tr>
<tr>
<td>$i_1$</td>
<td>$q_1$ $q_2$ $q_3$</td>
</tr>
</tbody>
</table>

The output signals, the only externally visible indication of the internal state (the light) are described by the table

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$o_0$ $o_0$ $o_1$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$o_0$ $o_0$ $o_1$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$o_0$ $o_0$ $o_1$</td>
</tr>
</tbody>
</table>

This example is typical of discrete-state machines. They can be described by such tables provided they have only a finite number of possible states.

It will seem that given the initial state of the machine and the input signals it is always possible to predict all future states. This is reminiscent of Laplace’s view that from the complete state of the universe at one moment of time, as described by the positions and velocities of all particles, it should be possible to predict all future states. The prediction which we are considering is, however, rather nearer to practicability than that considered by Laplace. The system of the “universe as a whole” is such that quite small errors in the initial conditions can have an overwhelming effect at a later time. The displacement of a single electron by a billionth of a centimetre at one moment might make the difference between a man being killed by an avalanche a year later, or escaping. It is an essential property of the mechanical systems which we have called “discrete-state machines” that this phenomenon does not occur. Even when we consider the actual physical machines instead of the idealised machines, reasonably accurate knowledge of the state at one moment yields reasonably accurate knowledge any number of steps later.
As we have mentioned, digital computers fall within the class of discrete-state machines. But the number of states of which such a machine is capable is usually enormously large. For instance, the number for the machine now working at Manchester is about $2^{165,000}$, i.e., about $10^{50,000}$. Compare this with our example of the clicking wheel described above, which had three states. It is not difficult to see why the number of states should be so immense. The computer includes a store corresponding to the paper used by a human computer. It must be possible to write into the store any one of the combinations of symbols which might have been written on the paper. For simplicity suppose that only digits from 0 to 9 are used as symbols. Variations in handwriting are ignored. Suppose the computer is allowed 100 sheets of paper each containing 50 lines each with room for 30 digits. Then the number of states is $10^{100 \times 50 \times 30}$, i.e., $10^{150,000}$. This is about the number of states of three Manchester machines put together. The logarithm to the base two of the number of states is usually called the “storage capacity” of the machine. Thus the Manchester machine has a storage capacity of about 165,000 and the wheel machine of our example about 1.6. If two machines are put together their capacities must be added to obtain the capacity of the resultant machine. This leads to the possibility of statements such as “The Manchester machine contains 64 magnetic tracks each with a capacity of 2560, eight electronic tubes with a capacity of 1280. Miscellaneous storage amounts to about 300 making a total of 174,380.”

Given the table corresponding to a discrete-state machine it is possible to predict what it will do. There is no reason why this calculation should not be carried out by means of a digital computer. Provided it could be carried out sufficiently quickly the digital computer could mimic the behavior of any discrete-state machine. The imitation game could then be played with the machine in question (as B) and the mimicking digital computer (as A) and the interrogator would be unable to distinguish them. Of course the digital computer must have an adequate storage capacity as well as working sufficiently fast. Moreover, it must be programmed afresh for each new machine which it is desired to mimic.

This special property of digital computers, that they can mimic any discrete-state machine, is described by saying that they are universal machines. The existence of machines with this property has the important consequence that, considerations of speed apart, it is unnecessary to design various new machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each case. It will be seen that as a consequence of this all digital computers are in a sense equivalent.

We may now consider again the point raised at the end of §3. It was suggested tentatively that the question, “Can machines think?” should be replaced by “Are there imaginable digital computers which would do well in the imitation game?” If we wish we can make this superficially more general and ask “Are there discrete-state machines which would do well?” But in view of the universality property we see that either of these questions is equivalent to this, “Let us fix our attention on one particular digital computer C. Is it true that by modifying this computer to have an adequate storage, suitably increasing its speed of action, and providing it with an appropriate programme, C can be made to play satisfactorily the part of A in the imitation game, the part of B being taken by a man?”

6. Contrary Views on the Main Question

We may now consider the ground to have been cleared and we are ready to proceed to the debate on our question, “Can machines think?” and the variant of it quoted at the end of the last section. We cannot altogether abandon the original form of the problem, for opinions will differ as to the appropriateness of the substitution and we must at least listen to what has to be said in this connexion.

It will simplify matters for the reader if I explain first my own beliefs in the matter. Consider first the more accurate form of the question. I believe that in about fifty years’ time it will be possible, to programme computers, with a storage capacity of about $10^9$, to make them play the imitation game so well that an average interrogator will not have more than 70 per cent chance of making the right identification after five minutes of questioning. The original question, “Can machines think?” I believe to be too
meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted. I believe further that no useful purpose is served by concealing these beliefs. The popular view that scientists proceed inexorably from well-established fact to well-established fact, never being influenced by any improved conjecture, is quite mistaken. Provided it is made clear which are proved facts and which are conjectures, no harm can result. Conjectures are of great importance since they suggest useful lines of research.

I now proceed to consider opinions opposed to my own.

(1) The Theological Objection. Thinking is a function of man’s immortal soul. God has given an immortal soul to every man and woman, but not to any other animal or to machines. Hence no animal or machine can think.

I am unable to accept any part of this, but will attempt to reply in theological terms. I should find the argument more convincing if animals were classed with men, for there is a greater difference, to my mind, between the typical animate and the inanimate than there is between man and the other animals. The arbitrary character of the orthodox view becomes clearer if we consider how it might appear to a member of some other religious community. How do Christians regard the Moslem view that women have no souls? But let us leave this point aside and return to the main argument. It appears to me that the argument quoted above implies a serious restriction of the omnipotence of the Almighty. It is admitted that there are certain things that He cannot do such as making one equal to two, but should we not believe that He has freedom to confer a soul on an elephant if He sees fit? We might expect that He would only exercise this power in conjunction with a mutation which provided the elephant with an appropriately improved brain to minister to the needs of this sort. An argument of exactly similar form may be made for the case of machines. It may seem different because it is more difficult to “swallow.” But this really only means that we think it would be less likely that He would consider the circumstances suitable for conferring a soul. The circumstances in question are discussed in the rest of this paper. In attempting to construct such machines we should not be irreverently usurping His power of creating souls, any more than we are in the procreation of children: rather we are, in either case, instruments of His will providing mansions for the souls that He creates.1

However, this is mere speculation. I am not very impressed with theological arguments whatever they may be used to support. Such arguments have often been found unsatisfactory in the past. In the time of Galileo it was argued that the texts, “And the sun stood still . . . and hasted not to go down about a whole day” (Joshua x. 13) and “He laid the foundations of the earth, that it should not move at any time” (Psalm cv. 5) were an adequate refutation of the Copernican theory. With our present knowledge such an argument appears futile. When that knowledge was not available it made a quite different impression.

(2) The “Heads in the Sand” Objection. “The consequences of machines thinking would be too dreadful. Let us hope and believe that they cannot do so.”

This argument is seldom expressed quite so openly as in the form above. But it affects most of us who think about it at all. We like to believe that Man is in some subtle way superior to the rest of creation. It is best if he can be shown to be necessarily superior, for then there is no danger of him losing his commanding position. The popularity of the theological argument is clearly connected with this feeling. It

---

1 Possibly this view is heretical. St. Thomas Aquinas (Summa Theologica, quoted by Bertrand Russell, p. 480) states that God cannot make a man to have no soul. But this may not be a real restriction on His powers, but only a result of the fact that men’s souls are immortal, and therefore indestructible.
is likely to be quite strong in intellectual people, since they value the power of thinking more highly than others, and are more inclined to base their belief in the superiority of Man on this power.

I do not think that this argument is sufficiently substantial to require refutation. Consolation would be more appropriate: perhaps this should be sought in the transmigration of souls.

(3) The Mathematical Objection. There are a number of results of mathematical logic which can be used to show that there are limitations to the powers of discrete-state machines. The best known of these results is known as Gödel’s theorem (1931) and shows that in any sufficiently powerful logical system statements can be formulated which can neither be proved nor disproved within the system, unless possibly the system itself is inconsistent. There are other, in some respects similar, results due to Church (1936), Kleene (1935), Rosser, and Turing (1937). The latter result is the most convenient to consider, since it refers directly to machines, whereas the others can only be used in a comparatively indirect argument: for instance if Gödel’s theorem is to be used we need in addition to have some means of describing logical systems in terms of machines, and machines in terms of logical systems. The result in question refers to a type of machine which is essentially a digital computer with an infinite capacity. It states that there are certain things that such a machine cannot do. If it is rigged up to give answers to questions as in the imitation game, there will be some questions to which it will either give a wrong answer, or fail to give an answer at all however much time is allowed for a reply. There may, of course, be many such questions, and questions which cannot be answered by one machine may be satisfactorily answered by another. We are of course supposing for the present that the questions are of the kind to which an answer “Yes” or “No” is appropriate, rather than questions such as “What do you think of Picasso?” The questions that we know the machines must fail on are of this type, “Consider the machine specified as follows. . . . Will this machine ever answer ‘Yes’ to any question?” The dots are to be replaced by a description of some machine in a standard form, which could be something like that used in §5. When the machine described bears a certain comparatively simple relation to the machine which is under interrogation, it can be shown that the answer is either wrong or not forthcoming. This is the mathematical result: it is argued that it proves a disability of machines to which the human intellect is not subject.

The short answer to this argument is that although it is established that there are limitations to the Powers If any particular machine, it has only been stated, without any sort of proof, that no such limitations apply to the human intellect. But I do not think this view can be dismissed quite so lightly. Whenever one of these machines is asked the appropriate critical question, and gives a definite answer, we know that this answer must be wrong, and this gives us a certain feeling of superiority. Is this feeling illusory? It is no doubt quite genuine, but I do not think too much importance should be attached to it. We too often give wrong answers to questions ourselves to be justified in being very pleased at such evidence of fallibility on the part of the machines. Further, our superiority can only be felt on such an occasion in relation to the one machine over which we have scored our petty triumph. There would be no question of triumphing simultaneously over all machines. In short, then, there might be men cleverer than any given machine, but then again there might be other machines cleverer again, and so on.

Those who hold to the mathematical argument would, I think, mostly he willing to accept the imitation game as a basis for discussion. Those who believe in the two previous objections would probably not be interested in any criteria.

(4) The Argument from Consciousness. This argument is very, well expressed in Professor Jefferson’s Lister Oration for 1949, from which I quote. “Not until a machine can write a sonnet or compose a concerto because of thoughts and emotions felt, and not by the chance fall of symbols, could we agree that machine equals brain—that is, not only write it but know that it had written it. No mechanism could feel (and not merely artificially signal, an easy contrivance) pleasure at its successes, grief when its valves fuse, be
warmed by flattery, be made miserable by its mistakes, be charmed by sex, be angry or depressed when it
cannot get what it wants.”

This argument appears to be a denial of the validity of our test. According to the most extreme form of
this view the only way by which one could be sure that machine thinks is to be the machine and to feel
oneself thinking. One could then describe these feelings to the world, but of course no one would be
justified in taking any notice. Likewise according to this view the only way to know that a man thinks is to
be that particular man. It is in fact the solipsist point of view. It may be the most logical view to hold but it
makes communication of ideas difficult. A is liable to believe “A thinks but B does not” whilst B believes
“B thinks but A does not.” instead of arguing continually over this point it is usual to have the polite
convention that everyone thinks.

I am sure that Professor Jefferson does not wish to adopt the extreme and solipsist point of view.
Probably he would be quite willing to accept the imitation game as a test. The game (with the player B
omitted) is frequently used in practice under the name of viva voce to discover whether some one really
understands something or has “learnt it parrot fashion.” Let us listen in to a part of such a viva voce:

Interrogator: In the first line of your sonnet which reads “Shall I compare thee to a summer’s day,”
would not “a spring day” do as well or better?
Witness: It wouldn’t scan.
Interrogator: How about “a winter’s day;” That would scan all right.
Witness: Yes, but nobody wants to be compared to a winter’s day.
Interrogator: Would you say Mr. Pickwick reminded you of Christmas?
Witness: In a way.
Interrogator: Yet Christmas is a winter’s day, and I do not think Mr. Pickwick would mind the
comparison.
Witness: I don’t think you’re serious. By a winter’s day one means a typical winter’s day, rather than a
special one like Christmas.

And so on. What would Professor Jefferson say if the sonnet-writing machine was able to answer like
this in the viva voce? I do not know whether he would regard the machine as “merely artificially signalling”
these answers, but if the answers were as satisfactory and sustained as in the above passage I do not think
he would describe it as “an easy contrivance.” This phrase is, I think, intended to cover such devices as the
inclusion in the machine of a record of someone reading a sonnet, with appropriate switching to turn it on
from time to time.

In short then, I think that most of those who support the argument from consciousness could be
persuaded to abandon it rather than be forced into the solipsist position. They will then probably be willing
to accept our test.

I do not wish to give the impression that I think there is no mystery about consciousness. There is, for
instance, something of a paradox connected with any attempt to localise it. But I do not think these
mysteries necessarily need to be solved before we can answer the question with which we are concerned in
this paper.

(5) Arguments from Various Disabilities. These arguments take the form, “I grant you that you can make
machines do all the things you have mentioned but you will never be able to make one to do X.”
Numerous features X are suggested in this connexion I offer a selection:
Be kind, resourceful, beautiful, friendly, have initiative, have a sense of humour, tell right from wrong, make mistakes, fall in love, enjoy strawberries and cream, make some one fall in love with it, learn from experience, use words properly, be the subject of its own thought, have as much diversity of behaviour as a man, do something really new.

No support is usually offered for these statements. I believe they are mostly founded on the principle of scientific induction. A man has seen thousands of machines in his lifetime. From what he sees of them he draws a number of general conclusions. They are ugly, each is designed for a very limited purpose, when required for a minutely different purpose they are useless, the variety of behaviour of any one of them is very small, etc., etc. Naturally he concludes that these are necessary properties of machines in general. Many of these limitations are associated with the very small storage capacity of most machines. (I am assuming that the idea of storage capacity is extended in some way to cover machines other than discrete-state machines. The exact definition does not matter as no mathematical accuracy is claimed in the present discussion.) A few years ago, when very little had been heard of digital computers, it was possible to elicit much incredulity concerning them, if one mentioned their properties without describing their construction. That was presumably due to a similar application of the principle of scientific induction. These applications of the principle are of course largely unconscious. When a burnt child fears the fire and shows that he fears it by avoiding it, f should say that he was applying scientific induction. (I could of course also describe his behaviour in many other ways.) The works and customs of mankind do not seem to be very suitable material to which to apply scientific induction. A very large part of space-time must be investigated, if reliable results are to be obtained. Otherwise we may (as most English children do) decide that everybody speaks English, and that it is silly to learn French.

There are, however, special remarks to be made about many of the disabilities that have been mentioned. The inability to enjoy strawberries and cream may have struck the reader as frivolous. Possibly a machine might be made to enjoy this delicious dish, but any attempt to make one do so would be idiotic. What is important about this disability is that it contributes to some of the other disabilities, e.g., to the difficulty of the same kind of friendliness occurring between man and machine as between white man and white man, or between black man and black man.

The claim that “machines cannot make mistakes” seems a curious one. One is tempted to retort, “Are they any the worse for that?” But let us adopt a more sympathetic attitude, and try to see what is really meant. I think this criticism can be explained in terms of the imitation game. It is claimed that the interrogator could distinguish the machine from the man simply by setting them a number of problems in arithmetic. The machine would be unmasked because of its deadly accuracy. The reply to this is simple. The machine (programmed for playing the game) would not attempt to give the right answers to the arithmetic problems. It would deliberately introduce mistakes in a manner calculated to confuse the interrogator. A mechanical fault would probably show itself through an unsuitable decision as to what sort of a mistake to make in the arithmetic. Even this interpretation of the criticism is not sufficiently sympathetic. But we cannot afford the space to go into it much further. It seems to me that this criticism depends on a confusion between two kinds of mistake. We may call them “errors of functioning” and “errors of conclusion.” Errors of functioning are due to some mechanical or electrical fault which causes the machine to behave otherwise than it was designed to do. In philosophical discussions one likes to ignore the possibility of such errors; one is therefore discussing “abstract machines.” These abstract machines are mathematical fictions rather than physical objects. By definition they are incapable of errors of functioning. In this sense we can truly say that “machines can never make mistakes.” Errors of conclusion can only arise when some meaning is attached to the output signals from the machine. The machine might, for instance, type out mathematical equations, or sentences in English. When a false proposition is typed we say that the machine has committed an error of conclusion. There is clearly no reason at all for saying that a machine cannot make this kind of mistake. It might do nothing but type out
repeatedly “0 = 1.” To take a less perverse example, it might have some method for drawing conclusions by scientific induction. We must expect such a method to lead occasionally to erroneous results.

The claim that a machine cannot be the subject of its own thought can of course only be answered if it can be shown that the machine has some thought with some subject matter. Nevertheless, “the subject matter of a machine’s operations” does seem to mean something, at least to the people who deal with it. If, for instance, the machine was trying to find a solution of the equation \( x^2 - 40x - 11 = 0 \) one would be tempted to describe this equation as part of the machine’s subject matter at that moment. In this sort of sense a machine undoubtedly can be its own subject matter. It may be used to help in making up its own programmes, or to predict the effect of alterations in its own structure. By observing the results of its own behaviour it can modify its own programmes so as to achieve some purpose more effectively. These are possibilities of the near future, rather than Utopian dreams.

The criticism that a machine cannot have much diversity of behaviour is just a way of saying that it cannot have much storage capacity. Until fairly recently a storage capacity of even a thousand digits was very rare.

The criticisms that we are considering here are often disguised forms of the argument from consciousness, Usually if one maintains that a machine can do one of these things, and describes the kind of method that the machine could use, one will not make much of an impression. It is thought that that method (whatever it may be, for it must be mechanical) is really rather base. Compare the parentheses in Jefferson’s statement quoted on page 22.

(6) Lady Lovelace’s Objection. Our most detailed information of Babbage’s Analytical Engine comes from a memoir by Lady Lovelace (1842). In it she states, “The Analytical Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform” (her italics). This statement is quoted by Hartree (1949) who adds: “This does not imply that it may not be possible to construct electronic equipment which will ‘think for itself,’ or in which, in biological terms, one could set up a conditioned reflex, which would serve as a basis for ‘learning.’ Whether this is possible in principle or not is a stimulating and exciting question, suggested by some of these recent developments. But it did not seem that the machines constructed or projected at the time had this property.”

I am in thorough agreement with Hartree over this. It will be noticed that he does not assert that the machines in question had not got the property, but rather that the evidence available to Lady Lovelace did not encourage her to believe that they had it. It is quite possible that the machines in question had in a sense got this property. For suppose that some discrete-state machine has the property. The Analytical Engine was a universal digital computer, so that, if its storage capacity and speed were adequate, it could by suitable programming be made to mimic the machine in question. Probably this argument did not occur to the Countess or to Babbage. In any case there was no obligation on them to claim all that could be claimed.

This whole question will be considered again under the heading of learning machines.

A variant of Lady Lovelace’s objection states that a machine can “never do anything really new.” This may be parried for a moment with the saw, “There is nothing new under the sun.” Who can be certain that “original work” that he has done was not simply the growth of the seed planted in him by teaching, or the effect of following well-known general principles. A better variant of the objection says that a machine can never “take us by surprise.” This statement is a more direct challenge and can be met directly. Machines take me by surprise with great frequency. This is largely because I do not do sufficient calculation to decide what to expect them to do, or rather because, although I do a calculation, I do it in a hurried, slipshod fashion, taking risks. Perhaps I say to myself, “I suppose the Voltage here ought to he the same as there: anyway let’s assume it is.” Naturally I am often wrong, and the result is a surprise for me for by the time the experiment is done these assumptions have been forgotten. These admissions lay me open to lectures on
the subject of my vicious ways, but do not throw any doubt on my credibility when I testify to the surprises I experience.

I do not expect this reply to silence my critic. He will probably say that the surprises are due to some creative mental act on my part, and reflect no credit on the machine. This leads us back to the argument from consciousness, and far from the idea of surprise. It is a line of argument we must consider closed, but it is perhaps worth remarking that the appreciation of something as surprising requires as much of a “creative mental act” whether the surprising event originates from a man, a book, a machine or anything else.

The view that machines cannot give rise to surprises is due, I believe, to a fallacy to which philosophers and mathematicians are particularly subject. This is the assumption that as soon as a fact is presented to a mind all consequences of that fact spring into the mind simultaneously with it. It is a very useful assumption under many circumstances, but one too easily forgets that it is false. A natural consequence of doing so is that one then assumes that there is no virtue in the mere working out of consequences from data and general principles.

(7) Argument from Continuity in the Nervous System. The nervous system is certainly not a discrete-state machine. A small error in the information about the size of a nervous impulse impinging on a neuron, may make a large difference to the size of the outgoing impulse. It may be argued that, this being so, one cannot expect to be able to mimic the behaviour of the nervous system with a discrete-state system.

It is true that a discrete-state machine must be different from a continuous machine. But if we adhere to the conditions of the imitation game, the interrogator will not be able to take any advantage of this difference. The situation can be made clearer if we consider some other simpler continuous machine. A differential analyser will do very well. (A differential analyser is a certain kind of machine not of the discrete-state type used for some kinds of calculation.) Some of these provide their answers in a typed form, and so are suitable for taking part in the game. It would not be possible for a digital computer to predict exactly what answers the differential analyser would give to a problem, but it would be quite capable of giving the right sort of answer. For instance, if asked to give the value of (actually about 3.1416) it would be reasonable to choose at random between the values 3.12, 3.13, 3.14, 3.15, 3.16 with the probabilities of 0.05, 0.15, 0.55, 0.19, 0.06 (say). Under these circumstances it would be very difficult for the interrogator to distinguish the differential analyser from the digital computer.

(8) The Argument from Informality of Behaviour. It is not possible to produce a set of rules purporting to describe what a man should do in every conceivable set of circumstances. One might for instance have a rule that one is to stop when one sees a red traffic light, and to go if one sees a green one, but what if by some fault both appear together? One may perhaps decide that it is safest to stop. But some further difficulty may well arise from this decision later. To attempt to provide rules of conduct to cover every eventuality, even those arising from traffic lights, appears to be impossible. With all this I agree.

From this it is argued that we cannot be machines. I shall try to reproduce the argument, but I fear I shall hardly do it justice. It seems to run something like this. “if each man had a definite set of rules of conduct by which he regulated his life he would be no better than a machine. But there are no such rules, so men cannot be machines.” The undistributed middle is glaring. I do not think the argument is ever put quite like this, but I believe this is the argument used nevertheless. There may however be a certain confusion between “rules of conduct” and “laws of behaviour” to cloud the issue. By “rules of conduct” I mean precepts such as “Stop if you see red lights,” on which one can act, and of which one can be conscious. By “laws of behaviour” I mean laws of nature as applied to a man’s body such as “if you pinch him he will squeak.” If we substitute “laws of behaviour which regulate his life” for “laws of conduct by which he regulates his life” in the argument quoted the undistributed middle is no longer insuperable. For we believe
that it is not only true that being regulated by laws of behaviour implies being some sort of machine (though not necessarily a discrete-state machine), but that conversely being such a machine implies being regulated by such laws. However, we cannot so easily convince ourselves of the absence of complete laws of behaviour as of complete rules of conduct. The only way we know of for finding such laws is scientific observation, and we certainly know of no circumstances under which we could say, “We have searched enough. There are no such laws.”

We can demonstrate more forcibly that any such statement would be unjustified. For suppose we could be sure of finding such laws if they existed. Then given a discrete-state machine it should certainly be possible to discover by observation sufficient about it to predict its future behaviour, and this within a reasonable time, say a thousand years. But this does not seem to be the case. I have set up on the Manchester computer a small programme using only 1,000 units of storage, whereby the machine supplied with one sixteen-figure number replies with another within two seconds. I would defy anyone to learn from these replies sufficient about the programme to be able to predict any replies to untried values.

(9) The Argument from Extra-Sensory Perception. I assume that the reader is familiar with the idea of extra-sensory perception, and the meaning of the four items of it, viz., telepathy, clairvoyance, precognition and psycho-kinesis. These disturbing phenomena seem to deny all our usual scientific ideas. How we should like to discredit them! Unfortunately the statistical evidence, at least for telepathy, is overwhelming. It is very difficult to rearrange one’s ideas so as to fit these new facts in. Once one has accepted them it does not seem a very big step to believe in ghosts and bogies. The idea that our bodies move simply according to the known laws of physics, together with some others not yet discovered but somewhat similar, would be one of the first to go.

This argument is to my mind quite a strong one. One can say in reply that many scientific theories seem to remain workable in practice, in spite of clashing with E.S.P.; that in fact one can get along very nicely if one forgets about it. This is rather cold comfort, and one fears that thinking is just the kind of phenomenon where E.S.P. may be especially relevant.

A more specific argument based on E.S.P. might run as follows: “Let us play the imitation game, using as witnesses a man who is good as a telepathic receiver, and a digital computer. The interrogator can ask such questions as ‘What suit does the card in my right hand belong to?’ The man by telepathy or clairvoyance gives the right answer 130 times out of 400 cards. The machine can only guess at random, and perhaps gets 104 right, so the interrogator makes the right identification.” There is an interesting possibility which opens here. Suppose the digital computer contains a random number generator. Then it will be natural to use this to decide what answer to give. But then the random number generator will be subject to the psycho-kinetic powers of the interrogator. Perhaps this psycho-kinesis might cause the machine to guess right more often than would be expected on a probability calculation, so that the interrogator might still be unable to make the right identification. On the other hand, he might be able to guess right without any questioning, by clairvoyance. With E.S.P. anything may happen.

If telepathy is admitted it will be necessary to tighten our test up. The situation could be regarded as analogous to that which would occur if the interrogator were talking to himself and one of the competitors was listening with his ear to the wall. To put the competitors into a “telepathy-proof room” would satisfy all requirements.

7. Learning Machines

The reader will have anticipated that I have no very convincing arguments of a positive nature to support my views. If I had I should not have taken such pains to point out the fallacies in contrary views. Such evidence as I have I shall now give.
Let us return for a moment to Lady Lovelace’s objection, which stated that the machine can only do what we tell it to do. One could say that a man can “inject” an idea into the machine, and that it will respond to a certain extent and then drop into quiescence, like a piano string struck by a hammer. Another simile would be an atomic pile of less than critical size: an injected idea is to correspond to a neutron entering the pile from without. Each such neutron will cause a certain disturbance which eventually dies away. If, however, the size of the pile is sufficiently increased, the disturbance caused by such an incoming neutron will very likely go on and on increasing until the whole pile is destroyed. Is there a corresponding phenomenon for minds, and is there one for machines? There does seem to be one for the human mind. The majority of them seem to be “subcritical,” i.e., to correspond in this analogy to piles of subcritical size. An idea presented to such a mind will on average give rise to less than one idea in reply. A smallish proportion are supercritical. An idea presented to such a mind that may give rise to a whole “theory” consisting of secondary, tertiary and more remote ideas. Animals minds seem to be very definitely subcritical. Adhering to this analogy we ask, “Can a machine be made to be supercritical?”

The “skin-of-an-onion” analogy is also helpful. In considering the functions of the mind or the brain we find certain operations which we can explain in purely mechanical terms. This we say does not correspond to the real mind: it is a sort of skin which we must strip off if we are to find the real mind. But then in what remains we find a further skin to be stripped off, and so on. Proceeding in this way do we ever come to the “real” mind, or do we eventually come to the skin which has nothing in it? In the latter case the whole mind is mechanical. (It would not be a discrete-state machine however. We have discussed this.)

These last two paragraphs do not claim to be convincing arguments. They should rather be described as “recitations tending to produce belief.”

The only really satisfactory support that can be given for the view expressed at the beginning of §6, will be that provided by waiting for the end of the century and then doing the experiment described. But what can we say in the meantime? What steps should be taken now if the experiment is to be successful?

As I have explained, the problem is mainly one of programming. Advances in engineering will have to be made too, but it seems unlikely that these will not be adequate for the requirements. Estimates of the storage capacity of the brain vary from \(10^{10}\) to \(10^{15}\) binary digits. I incline to the lower values and believe that only a very small fraction is used for the higher types of thinking. Most of it is probably used for the retention of visual impressions, I should be surprised if more than \(10^9\) was required for satisfactory playing of the imitation game, at any rate against a blind man. (Note: The capacity of the Encyclopædia Britannica, 11th edition, is \(2 \times 10^9\) ) A storage capacity of \(10^7\), would be a very practicable possibility even by present techniques. It is probably not necessary to increase the speed of operations of the machines at all. Parts of modern machines which can be regarded as analogs of nerve cells work about a thousand times faster than the latter. This should provide a “margin of safety” which could cover losses of speed arising in many ways. Our problem then is to find out how to programme these machines to play the game. At my present rate of working I produce about a thousand digits of programme a day, so that about sixty workers, working steadily through the fifty years might accomplish the job, if nothing went into the wastepaper basket. Some more expeditious method seems desirable.

In the process of trying to imitate an adult human mind we are bound to think a good deal about the process which has brought it to the state that it is in. We may notice three components.

\(a\) The initial state of the mind, say at birth,

\(b\) The education to which it has been subjected,

\(c\) Other experience, not to be described as education, to which it has been subjected.
Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child’s? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child brain is something like a notebook as one buys it from the stationer’s. Rather little mechanism, and lots of blank sheets. (Mechanism and writing are from our point of view almost synonymous.) Our hope is that there is so little mechanism in the child brain that something like it can be easily programmed. The amount of work in the education we can assume, as a first approximation, to be much the same as for the human child.

We have thus divided our problem into two parts. The child programme and the education process. These two remain very closely connected. We cannot expect to find a good child machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications

\[
\begin{align*}
\text{Structure of the child machine} & = \text{hereditary material} \\
\text{Changes of the child machine} & = \text{mutation,} \\
\text{Natural selection} & = \text{judgment of the experimenter}
\end{align*}
\]

One may hope, however, that this process will be more expeditious than evolution. The survival of the fittest is a slow method for measuring advantages. The experimenter, by the exercise of intelligence, should he able to speed it up. Equally important is the fact that he is not restricted to random mutations. If he can trace a cause for some weakness he can probably think of the kind of mutation which will improve it.

It will not be possible to apply exactly the same teaching process to the machine as to a normal child. It will not, for instance, be provided with legs, so that it could not be asked to go out and fill the coal scuttle. Possibly it might not have eyes. But however well these deficiencies might be overcome by clever engineering, one could not send the creature to school without the other children making excessive fun of it. It must be given some tuition. We need not be too concerned about the legs, eyes, etc. The example of Miss Helen Keller shows that education can take place provided that communication in both directions between teacher and pupil can take place by some means or other.

We normally associate punishments and rewards with the teaching process. Some simple child machines can be constructed or programmed on this sort of principle. The machine has to be so constructed that events which shortly preceded the occurrence of a punishment signal are unlikely to be repeated, whereas a reward signal increased the probability of repetition of the events which led up to it. These definitions do not presuppose any feelings on the part of the machine, I have done some experiments with one such child machine, and succeeded in teaching it a few things, but the teaching method was too unorthodox for the experiment to be considered really successful.

The use of punishments and rewards can at best be a part of the teaching process. Roughly speaking, if the teacher has no other means of communicating to the pupil, the amount of information which can reach him does not exceed the total number of rewards and punishments applied. By the time a child has learnt to repeat “Casabianca” he would probably feel very sore indeed, if the text could only be discovered by a “Twenty Questions” technique, every “NO” taking the form of a blow. It is necessary therefore to have some other “unemotional” channels of communication. If these are available it is possible to teach a machine by punishments and rewards to obey orders given in some language, e.g., a symbolic language. These orders are to be transmitted through the “unemotional” channels. The use of this language will diminish greatly the number of punishments and rewards required.

Opinions may vary as to the complexity which is suitable in the child machine. One might try to make it as simple as possible consistently with the general principles. Alternatively one might have a complete system of logical inference “built in.” In the latter case the store would be largely occupied with definitions
and propositions. The propositions would have various kinds of status, e.g., well-established facts, conjectures, mathematically proved theorems, statements given by an authority, expressions having the logical form of proposition but not belief-value. Certain propositions may be described as “imperatives.” The machine should be so constructed that as soon as an imperative is classed as “well established” the appropriate action automatically takes place. To illustrate this, suppose the teacher says to the machine, “Do your homework now.” This may cause “Teacher says ‘Do your homework now’” to be included amongst the well-established facts. Another such fact might be, “Everything that teacher says is true.” Combining these may eventually lead to the imperative, “Do your homework now,” being included amongst the well-established facts, and this, by the construction of the machine, will mean that the homework actually gets started, but the effect is very satisfactory. The processes of inference used by the machine need not be such as would satisfy the most exacting logicians. There might for instance be no hierarchy of types. But this need not mean that type fallacies will occur, any more than we are bound to fall over unfenced cliffs. Suitable imperatives (expressed within the systems, not forming part of the rules of the system) such as “Do not use a class unless it is a subclass of one which has been mentioned by teacher” can have a similar effect to “Do not go too near the edge.”

The imperatives that can be obeyed by a machine that has no limbs are bound to be of a rather intellectual character, as in the example (doing homework) given above. Important amongst such imperatives will be ones which regulate the order in which the rules of the logical system concerned are to be applied. For at each stage when one is using a logical system, there is a very large number of alternative steps, any of which one is permitted to apply, so far as obedience to the rules of the logical system is concerned. These choices make the difference between a brilliant and a footling reasoner, not the difference between a sound and a fallacious one. Propositions leading to imperatives of this kind might be “When Socrates is mentioned, use the syllogism in Barbara” or “If one method has been proved to be quicker than another, do not use the slower method.” Some of these may be “given by authority,” but others may be produced by the machine itself, e.g. by scientific induction.

The idea of a learning machine may appear paradoxical to some readers. How can the rules of operation of the machine change? They should describe completely how the machine will react whatever its history might be, whatever changes it might undergo. The rules are thus quite time-invariant. This is quite true. The explanation of the paradox is that the rules which get changed in the learning process are of a rather less pretentious kind, claiming only an ephemeral validity. The reader may draw a parallel with the Constitution of the United States.

An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil’s behavior. This should apply most strongly to the later education of a machine arising from a child machine of well-tried design (or programme). This is in clear contrast with normal procedure when using a machine to do computations one’s object is then to have a clear mental picture of the state of the machine at each moment in the computation. This object can only be achieved with a struggle. The view that “the machine can only do what we know how to order it to do,” appears strange in face of this. Most of the programmes which we can put into the machine will result in its doing something that we cannot make sense (if at all, or which we regard as completely random behaviour. Intelligent behaviour presumably consists in a departure from the completely disciplined behaviour involved in computation, but a rather slight one, which does not give rise to random behaviour, or to pointless repetitive loops. Another important result of preparing our machine for its part in the imitation game by a process of teaching and learning is that “human fallibility” is likely to be omitted in a rather natural way, i.e., without special “coaching.” (The reader should reconcile this with the point of view on pages 23 and 24.) Processes that are learnt do not produce a hundred per cent certainty of result; if they did they could not be unlearnt.
It is probably wise to include a random element in a learning machine. A random element is rather useful when we are searching for a solution of some problem. Suppose for instance we wanted to find a number between 50 and 200 which was equal to the square of the sum of its digits, we might start at 51 then try 52 and go on until we got a number that worked. Alternatively we might choose numbers at random until we got a good one. This method has the advantage that it is unnecessary to keep track of the values that have been tried, but the disadvantage that one may try the same one twice, but this is not very important if there are several solutions. The systematic method has the disadvantage that there may be an enormous block without any solutions in the region which has to be investigated first, Now the learning process may be regarded as a search for a form of behaviour which will satisfy the teacher (or some other criterion). Since there is probably a very large number of satisfactory solutions the random method seems to be better than the systematic. It should be noticed that it is used in the analogous process of evolution. But there the systematic method is not possible. How could one keep track of the different genetical combinations that had been tried, so as to avoid trying them again?

We may hope that machines will eventually compete with men in all purely intellectual fields. But which are the best ones to start with? Even this is a difficult decision. Many people think that a very abstract activity, like the playing of chess, would be best. It can also be maintained that it is best to provide the machine with the best sense organs that money can buy, and then teach it to understand and speak English. This process could follow the normal teaching of a child. Things would be pointed out and named, etc. Again I do not know what the right answer is, but I think both approaches should be tried.

We can only see a short distance ahead, but we can see plenty there that needs to be done.

BIBLIOGRAPHY


Countess of Lovelace, “Translator’s notes to an article on Babbage’s Analytical Engine,” *Scientific Memoirs* (ed. by R. Taylor), vol. 3 (1842), 691-731.


*Victoria University of Manchester.*