We use the term computing perspective to refer to a way of looking at the world as it would be seen through the eyes of a computer scientist. You do not have to be a computer scientist to do this, but you do need to know a handful of the fundamental concepts that help computer scientists see things in a certain way.

Computers are devices that do only one kind of thing: They carry out algorithms to process information. To computer scientists, the algorithm is the central unifying concept of computing, the mode of thought that is at the core of the computing perspective. Thus, to see things from the computing perspective, you need to understand algorithms.

In this chapter we introduce algorithms, show simple examples of them, explain their relationship to computers, and discuss the keys to seeing things from the computing perspective.

2.1 What Is an Algorithm?

Any precise set of instructions that adequately specifies behavior and is not ambiguous, can be considered to be an algorithm.

**Definition:** An algorithm is a specification of a behavioral process. It consists of a finite set of instructions that govern behavior step-by-step.
Although algorithms are central to computing, they need not have anything at all to do with computers. Years before computers were invented, your great-grandmother carried out an algorithm every time she followed a recipe to make an apple pie. Other everyday examples of algorithms abound.

EXAMPLE 2.1 When I assembled the office chair in which I am now sitting, I executed the following algorithm that was provided by the manufacturer:

algorithm 'Assembly Instructions for Office Chair' 
1. Place the chair seat upside down on a table or other flat work surface.
2. Attach the mounting plate to the underside of the chair seat using 4 bolts.
3. Attach the chair arms to the underside of the chair seat using 2 bolts per arm.
4. Place the chair base with castors on the floor.
5. Insert the large end of the pneumatic cylinder into the center of the chair base.
6. Slide the plastic cover over the pneumatic cylinder.
7. Place the chair seat (with attached mounting plate and arms) over the small end of the pneumatic cylinder. Guide the top end of the cylinder into the hole in the mounting plate. Press down firmly on the chair seat.
8. Install the backrest by sliding its metal prong into the slot at the rear of the mounting plate.
9. Adjust the backrest to a comfortable height, then tighten the knob at the rear of the mounting plate.
end algorithm

Algorithms for tasks such as baking pies or assembling office chairs specify behavioral steps that involve the physical manipulation of physical matter (e.g., cooking ingredients or assembling chair components). In contrast, using algorithms and computers to perform tasks involves the manipulation of data, simply because data processing is what computers do. We will discuss the relationship of algorithms to computers later in this chapter.

Any algorithm receives input, processes that input according to its specification of behavior, and produces output. In the case of the algorithm for assembling an office chair, the input is the set of unassembled chair parts, the algorithm is the list of assembly instructions, and the output is the assembled chair. For algorithms that process data, the same model applies:

![Input data → Algorithm → Output data](image)

Before we examine the requirements of algorithms, let's look at an algorithm that solves a data-processing problem.

EXAMPLE 2.2 Problem specification: Process a list of the baseball players in a league and their performance at the end of the season to determine the league’s batting champion for that year. A player must have a minimum number of “at bats” (i.e.,
attempts) in order to be eligible. The player with highest batting average (number of
hits divided by number of at bats) is the batting champion.

Context: The processor of this list should be a person using pencil and paper, not
a computer. When we say “create a space” or “record some number,” the task should
be done on paper.

Attempted solution:

algorithm "Batting Champion"
1. Create labeled space for a number called "MIN AB."
2. Obtain the minimum number of at bats required.
3. Record that number as "MIN AB."
4. Obtain a master list of baseball players that includes the needed perfor-
   mance data: player’s name (Name), number of hits (Hits), and number
   of at bats (AB).
5. Create labeled space for a name called "BA Leader."
6. Create labeled space for a number called "Best BA."
7. Begin with a zero average: record 0 as "Best BA."
8. Consider the data for the first player in the master list.
9. If (this player’s AB is greater than MIN AB) then
   9.1. Create labeled space for a number called "This BA."
   9.2. Divide this player’s Hits by his AB, record the result as
        "This BA."
   9.3. If ("This BA" is greater than "Best BA") then
      9.3.1. Record "This BA" as "Best BA."
      9.3.2. Record this player’s Name as "BA Leader."
10. Cross this player off the master list and consider the data for the
    next player in the list.
11. Repeat steps 9 and 10 until every player in the master list has been
    considered.
12. Create labeled space for a name called "Batting Champion."
13. Record the name in "BA Leader" as "Batting Champion."
end algorithm.

Notice how indentation is used to show that steps 9.1 through 9.3 are executed only
if the condition in line 9 is true. If the condition in line 9 is not true, then 9.1, 9.2,
and 9.3 are not executed. Similarly, lines 9.3.1 and 9.3.2 are indented under line 9.3,
indicating that they will be executed only if the condition in line 9.3 is true.

The attempted solution is indeed an algorithm, in that it consists of a finite set of
instructions that governs behavior step-by-step. But is it correct? At first glance, it
might appear to be. In fact, in most circumstances, it will correctly perform the task.
Unfortunately, it will not necessarily produce correct results. There are two circum-
stances in which the solution will fail to produce correct results:

- In line 9 the algorithm checks to see if each player’s number of at bats is greater
  than the minimum required for the batting champion. If a given player’s num-
  ber of at bats is equal to the minimum, that player should qualify, but this algo-
  rithm misses him.
- In line 9.3 the algorithm checks to see if each player’s performance is greater
  than the best batting average so far. If several players have the highest, they
  should share the championship for that category. This algorithm recognizes only
  the one who appeared first in the list.
These two errors are of the same nature: \textit{The algorithm tests for the wrong condition}. It tests for "greater than" when it should test for "greater than or equal to." Notice, however, that these two errors have different consequences:

- The first error will produce results that are \textit{wrong}.
- The second error will produce results that are \textit{incomplete}.

In both cases, the errors occur because the algorithm failed to adequately consider all the possible cases that the data might be expected to present.

\textbf{Warning:} Successful algorithms must consider all possible cases presented by acceptable data.

From these examples of errors, we can specify another requirement of algorithms:

\textbf{Definition:} To be correct, an algorithm must produce results that are correct and complete given any and all sets of appropriate data.

\textbf{EXAMPLE 2.3} Let's now try to repair the Batting Champion algorithm:

algorithm "Batting Champion"
1. Create labeled space for a number called "MIN AB."
2. Obtain the minimum number of at bats required.
3. Record that number as "MIN AB."
4. Obtain a master list of baseball players that includes the needed performance data: player's name (Name), number of hits (Hits), and number of at bats (AB).
5. Create labeled space for a list of names called "BA Leader."
6. Create labeled space for a number called "Best BA."
7. Begin with a zero average: record 0 as "Best BA."
8. Consider the data for the first player in the master list.
9. If (this player's AB is greater than or equal to MIN AB) then
   9.1. Create labeled space for a number called "This BA."
   9.2. Divide this player's Hits by his number of AB, record the result as "This BA."
   9.3. If ("This BA" is greater than "Best BA") then
       9.3.1. Record "This BA" as "Best BA."
       9.3.2. Erase the list of "BA Leader."
       9.3.3. Add this player's name to the list "BA Leader."
   9.4. If ("This BA" is equal to "Best BA") then
       9.4.1. Add this player's name to the list "BA Leader."
10. Cross this player's name off the master list of players and consider the data for the next player in the list.
11. Repeat steps 9 and 10 until every player in the master list has been considered.
12. Create labeled space for a list of names called "Batting Champion."
13. Record the list of names "BA Leader" as "Batting Champion."
end algorithm.
The repairs to our original version involved several changes:

- Because multiple players might tie for leader status, we changed lines 5 and 12 so they instruct the processor to create space for “a list of names” rather than just “a name.”
- For the same reason, instructions 9.3.2, 9.3.3, 9.4.1, and 13 all refer to “list of names” rather than “a name.”
- We changed the test in line 9, from “greater than” to “greater than or equal to,” to recognize players who have exactly met, as well as those who exceeded, the minimum number of at bats.
- The test in our original 9.3 was an error because it failed to recognize a player whose batting average tied the current leader. However, simply changing this test from “greater than” to “greater than or equal to” would not have been adequate. Why? Because we need to take different actions depending on whether the current player ties the current leader or betters him. If he exceeds the best performance so far, we must record the current player’s batting average as the new standard and erase the former leaders in that category (9.3.1 and 9.3.2). If the current player equalled the best performance so far but did not exceed it, we do not do those things. In either case, we add his name to the list of leaders (9.3.3 and 9.4.1). Again, we use indentation to indicate which instructions are executed only if the conditions in previous instructions are satisfied.

Our repairs made the totality of step 9 more complex. We had to do more logic to handle not only the normal case (when there is only a single leader) but also the exceptional case (when multiple players might be tied for the leadership).

The kind of evolution we have seen here in developing our Batting Champion algorithm is typical. Often, people who are new to the challenge of constructing algorithms will originally consider only the most obvious case and then will have to wrestle with making repairs.

**Warning:** You will succeed more quickly at constructing algorithms if you make it a habit to think first about the problem and its data, and then to enumerate all the special cases that the algorithm must handle.

### 2.2 Properties of Good Algorithms

After repairs, our Batting Champion algorithm is a correct data-processing algorithm. It is in “plain English” and therefore suitable to be executed by a “human processor” who understands English and who has access to the appropriate data.

There are many real-world circumstances in which such “natural language algorithms” are critically important. For example, any nurse who works in a critical care environment (e.g., in a shock-trauma, intensive care, or cardiac care unit) must be able at times to function primarily as a processor who executes algorithms. The same is true of combat aircraft pilots. Such work demands life-and-death decisions that must be made rapidly, often when there is no warning and no time to think. Indeed, individuals holding these jobs are extensively trained and tested to be able to function
EXAMPLE 2.4 We can restructure the Batting Champion algorithm to reflect more levels of abstraction. We do so by hiding the many details inside modules that we then “invoke” (or “use” or “call”) whenever they are needed.

algorithm 'Batting Champion'
1. Invoke 'Define MIN Num of At Bats'
2. Obtain a master list of baseball players that includes the needed performance data: player's name (Name), number of hits (Hits), and number of at bats (AB).
3. Invoke 'Set Up Needed Data Space.'
4. Consider the data for the first player in the master list.
5. If (this player's AB is greater than or equal to MIN AB) then
   5.1. Invoke 'Calculate Batting Average.'
5.2. Invoke 'Consider Batting Average for the Championship.'
6. Cross this player's name off the master list of players and consider the data for the next player in the list.
7. Repeat steps 5 and 6 until every player in the master list has been considered.
8. Invoke 'Output Findings Re: Batting Champion.'
end algorithm.

This results in an algorithm that is easier to read and understand.

By hiding the details inside appropriate modules, we can understand the main ideas without being distracted. This is a key goal of using levels of abstraction:

- Each module represents an abstraction. The name of the module describes the idea that the module implements. The instructions hidden within the module specify how that abstraction is implemented.
- We can see what is being done (the idea) by reading the descriptive name of the module without having to pay attention to how it is being done.
- If we want to understand how it is being done, we can look inside the module to find out.

For example, if we want to know exactly how “Consider Batting Average for the Championship” is specified, we can consult that module. Here's what it might look like:

module Consider Batting Average for the Championship
1. If ('This BA' is greater than 'Best BA') then
   1.1. Record 'This BA' as 'Best BA.'
   1.2. Erase the list 'BA Leader.'
   1.3. Add this player's Name to the list 'BA Leader.'
2. If ('This BA' is equal to 'Best Average') then
   2.1. Add this player's Name to the list 'BA Leader.'
end module

This is the same logic that had been cluttering up the "main" algorithm. We haven't changed it; we've just relocated it to get it out of the way. To do so, we've created a module that represents an abstraction—that is, "what it means to consider a player's average for the Batting Championship."
Definition: To make use of levels of abstraction, you can create an abstraction for each idea by doing the following:

- Create a module for the idea.
- Place the group of instructions that implement that idea inside the module.
- Invoke that module whenever the instructions for that idea need to be carried out.

Doing things this way gives us an algorithm that features a certain structure. We have a main algorithm that is not cluttered up with details. Instead, it describes the high-level logic of the algorithm. It coordinates the work of the modules and decides when each module is needed. It does not do the detailed work of the algorithm: the modules do that. Thus the main algorithm is analogous to a manager or coordinator, and each module to a skilled worker who knows how to do a specific job.

There may be any number of levels of abstraction in an algorithm, depending on the problem at hand. Higher and lower levels of abstraction are relative terms. The highest level is the level of algorithm purpose—for example, "identifying the leading hitters." The next highest level contains the logical steps involved in carrying out that purpose. Below that level are the particular instructions, typically hidden inside modules, which specify how to carry out each of those higher-level logical steps. One module may invoke another module, and so on, for as many levels of depth as are necessitated by the problem. Remember: It is the problem that guides the development of the algorithm; some algorithms may have few levels of abstraction, while others will have many, depending on the problem.

Warning: All algorithms must be constructed to feature levels of abstraction.

The crucial benefits of levels of abstraction become increasingly clear:

- Levels of abstraction are the means by which we can create algorithms that instruct processors to do complex things. Without them, algorithms would be so cluttered up with details that it would be practically impossible to understand them.
- Levels of abstraction allow us to easily substitute one set of particulars for another—that is, we can plug in modules that are equivalent in their effect but achieve that effect in different ways. By isolating details in a modular design, we can readily make selective improvements without having to undertake a major redesign.

Levels of abstraction thus make the complex jobs of creating, correcting, and updating algorithms manageable.

2.3 Algorithms and Computers

Algorithms give us a way to specify behavior via instructions. Computers are devices that can carry out instructions to manipulate data. They are machines that transform data, not physical matter. Computer science creates the means by which we can externalize the human ability to manipulate data into the things we make.
By using algorithms with computers, we seek to mechanize mental behavior, to incorporate ideas and abstractions into the computer so that the computer can carry out ideas for us by doing the behavior we specify.

When we construct algorithms for a computer to execute, we face a frustrating irony: a computer is a remarkably stupid device. It has no common sense whatsoever, it cannot resolve ambiguity, it requires every step to be very precise, and it will always do exactly what you tell it to do even if you have (accidentally) told it to do the wrong thing.

**Definition:** A computer is an electronic device that manipulates two levels of electric current: high and low. It treats the high level as a "one," the low level as a "zero." The only thing a computer can do is manipulate numbers consisting of ones and zeros; it can store them in memory, retrieve them from memory, do arithmetic operations on them, and compare their values.

Any time a computer appears to do something smart, it is an illusion. All it can do is manipulate ones and zeros very rapidly. The illusion of smartness comes from the algorithm that the computer is executing. It is the algorithm that makes the computer appear smart.

### 2.3.1 What Computers Do

When we construct algorithms for a computer, we face a frustrating irony: the computer is a remarkably stupid device. It has no common sense whatsoever, it cannot resolve ambiguity, it requires every step to be very precise, and it will always do exactly what you tell it to do even if you have (accidentally) told it to do the wrong thing.

**Definition:** A computer is an electronic device that manipulates two levels of electric current: high and low. It treats the high level as a "one," the low level as a "zero." The only thing a computer can do is manipulate numbers consisting of ones and zeros; it can store them in memory, retrieve them from memory, do arithmetic operations on them, and compare their values.

Any time a computer appears to do something smart, it is an illusion. All it can do is manipulate ones and zeros very rapidly. The illusion of smartness comes from the algorithm that the computer is executing. It is the algorithm that makes the computer appear smart.

### 2.3.2 Computers and Binary Data

Because computers only understand ones and zeros, they require the use of the base₂ notation, which represents numbers using only those two numerals. The numbering system used by people is base₁₀, which represents numbers using the numerals "0" through "9." Numbers represented in base₂ notation are called binary numbers. Numbers represented in base₁₀ notation are called decimal numbers.
### Table 2.1

<table>
<thead>
<tr>
<th>base&lt;sub&gt;n&lt;/sub&gt;</th>
<th>Digit 8</th>
<th>Digit 7</th>
<th>Digit 6</th>
<th>Digit 5</th>
<th>Digit 4</th>
<th>Digit 3</th>
<th>Digit 2</th>
<th>Digit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)&lt;sup&gt;7&lt;/sup&gt;</td>
<td>10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>10&lt;sup&gt;6&lt;/sup&gt;</td>
<td>10&lt;sup&gt;5&lt;/sup&gt;</td>
<td>10&lt;sup&gt;4&lt;/sup&gt;</td>
<td>10&lt;sup&gt;3&lt;/sup&gt;</td>
<td>10&lt;sup&gt;2&lt;/sup&gt;</td>
<td>10&lt;sup&gt;1&lt;/sup&gt;</td>
<td>10&lt;sup&gt;0&lt;/sup&gt;</td>
</tr>
<tr>
<td>10,000,000</td>
<td>1,000,000</td>
<td>100,000</td>
<td>10,000</td>
<td>1,000</td>
<td>100</td>
<td>10</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base&lt;sub&gt;2&lt;/sub&gt; (binary)</th>
<th>Digit 8</th>
<th>Digit 7</th>
<th>Digit 6</th>
<th>Digit 5</th>
<th>Digit 4</th>
<th>Digit 3</th>
<th>Digit 2</th>
<th>Digit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2&lt;sup&gt;7&lt;/sup&gt;</td>
<td>2&lt;sup&gt;6&lt;/sup&gt;</td>
<td>2&lt;sup&gt;5&lt;/sup&gt;</td>
<td>2&lt;sup&gt;4&lt;/sup&gt;</td>
<td>2&lt;sup&gt;3&lt;/sup&gt;</td>
<td>2&lt;sup&gt;2&lt;/sup&gt;</td>
<td>2&lt;sup&gt;1&lt;/sup&gt;</td>
<td>2&lt;sup&gt;0&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Different numbering systems follow the same pattern. For any given number, each digit represents the base raised to some power. The rightmost column represents the "ones" column or (base)<sup>0</sup>. Then, moving leftward, the next column represents (base)<sup>1</sup> or the base itself. The third digit represents (base)<sup>2</sup>, the fourth digit represents (base)<sup>3</sup>, and so on, with each column representing the (base number)<sup>power</sup> (the meaning of the column to its right). Table 2.1 shows the meaning of the first eight digits for decimal numbers and binary numbers.

To read a number in either system, we simply sum the digits according to what each one represents. Thus, the meaning of "2041" in base<sub>10</sub> is determined by adding two thousands, zero hundreds, four tens, and one one. The meaning of "1101" in base<sub>2</sub> is determined by adding one eight, one four, zero twos, and one one, which is thirteen. The number eleven is represented in base<sub>10</sub> as "11" because eleven consists of one ten and one one. In base<sub>2</sub>, eleven is represented as "1011" because it consists of one eight, zero fours, one two, and one one.

As you know, to make multi-digit decimal numbers easier for people to read, the digits are grouped in chunks of three, with each chunk separated by a comma, for example, 207,625. To make binary numbers easier to read, the digits are grouped in chunks of four, with chunks separated by blank space, for example, 1001 0111 (the binary representation of the decimal number 151).

#### 2.3.3 Levels of Abstraction in Computers

If we want a computer to be able to carry out an algorithm, the algorithm must be expressed in ones and zeros—the only form the computer can understand. This simple requirement presents a difficulty because people do not think in terms of ones and zeros. They think in terms of ideas and meanings. If a computer can understand only ones and zeros, we require some way to bridge the gap between the level at which people think and the level at which computers function.

When digital computers were first invented in the 1940s, there was no way to bridge that gap. Therefore computer programmers had to translate their ideas into ones and zeros. Because this method is so difficult, tedious, and error prone, later
generations of computer researchers have developed a better approach. For a computer system to execute an algorithm for a human user of that system, four logical levels of abstraction now come into play:

1. **The Human User Level**: The user interacts with a piece of application software (a computer program such as a spreadsheet, word processor, web navigator, or operating system) in order to complete some task. The user must only understand how to use the program; he or she does not need to understand the algorithm that makes it work.

2. **The Algorithm Level**: The program being used is an implementation of an algorithm that was developed by another person or group. Generally, algorithms are expressed in some algorithm language or code that is more precise, less ambiguous, and more compact than a “natural language” such as English.

   A language for writing algorithms may be conceptual (such as the one to be used here) or it may be a specific programming language (such as Fortran, Pascal, C++, Java, etc.). The advantage of using a programming language is that the algorithm can then be executed on a computer (see “The Translation Level” below). The disadvantage of programming languages is that they are created for specific purposes—that is, there is no single one that is best. In addition, programming languages usually involve a large number of annoying technical details that can make writing good algorithms more difficult.

3. **The Translation Level**: For an algorithm to be executed by a computer, the algorithm must be expressed in the particular code of some programming language. Every programming language comes with special translation software—either a compiler or an interpreter, which translates the code from its human-understandable form (as written by a person in a programming language) to a computer-understandable form (a list of ones and zeros). Thus, to have a usable computer program, one must (1) write the algorithm in a programming language and (2) have that program translated by a compiler or interpreter into a form that the computer can understand.

4. **The Hardware Level**: The hardware manipulates the ones and zeros it is given to produce output based on the input. The result may be text or graphics written to a computer screen, the sending of a document to a printer, the recording of data on a disk for later use, etc.

At each level some algorithmic process is at work. The least precise level is the top one: people often define a task in imprecise ways. At each of the lower levels, the algorithmic process is explicit, precise, and unambiguous. Table 2.2 summarizes the activity at each level.

---

3 Compilers and interpreters do equivalent jobs. They take as input the source code (written in some programming language) and output object code that the computer understands. Compilers perform the translation all at once, producing an entire body of code in ones and zeros. Interpreters do it by translating a single line, telling the computer to execute it, then translating the next line, and so on.

4 There is sometimes an intermediate level between the programming language and the binary code (assembly language), in which case an extra layer of translation is involved. Don’t worry about this. This is purely technical matter that has nothing to do with the logic of the algorithm and is of no importance here.
The ones and zeros we see at the hardware level are also used to store computer programs and data. These are called files on a computer disk (or another storage medium such as a CD-ROM or magnetic tape). The contents of every file is a list of ones and zeros, which means that the computer cannot tell whether a file is a data file or a program file; it simply does whatever it is told. When we run a program on some data, the computer is simply acting on one set of 1s and 0s (the program), which tells it how to manipulate some other set of 1s and 0s (the input data) to produce yet another set of 1s and 0s (the output data).

The representation of algorithms and data by binary numbers is the process that transformed computers into useful tools.

Fortunately, we do not have to concern ourselves with the ones and zeros, or with the details of how we get from our high-level ideas down to low-level machine instructions. (For those who are curious, there are other computer science courses that cover the entire range of these issues.) Thanks to the accomplishments of earlier computer scientists who built the software tools that give us the benefits of these levels of abstraction, we can ignore the low-level complexities and focus on the crucial level: the specification of behavior and how to articulate such specifications as algorithms.

Henceforth, we shall be concerned with the top two levels: the human level and the algorithm level. Except on occasion, we will not pay attention to the levels below these two. For now, it is important that you obtain a foundation in algorithms. With such a foundation, specific programming languages will be easier to learn.