CHAPTER 6

Algorithms

The idea of an *algorithm* is of fundamental importance in computer science and discrete mathematics. Broadly speaking, an algorithm is a sequence of commands that, if followed, result in some desirable outcome. In this sense a recipe for baking a cake is an algorithm. If you follow the instructions you get a cake. A typical algorithm has what we call *input*, that is, material or data that the algorithm uses, and *output*, which is the end result of the algorithm. In following the recipe for a cake, the ingredients are the input. The recipe (algorithm) tells what to do with the ingredients, and the output is a cake.

For another example, the instructions for making an origami swan from a piece of paper is an algorithm. The input is the paper, the algorithm is a sequence of instructions telling how to fold the paper, and the output is a (paper) swan. Different input (in color, size, etc.) leads to different output.

To *run* or *execute* an algorithm means to apply it to input and obtain output. Running or executing the swan algorithm produces a swan as output. We freely use the words "input" and "output" as both nouns and a verbs. The algorithm *inputs* a piece if paper and *outputs* a swan.

Today the word "algorithm" almost always refers to a sequence of steps written in a computer language and executed by a computer, and the input and output are information or data. Doing a Google search causes an algorithm to run. The "Google Algorithm" takes as input a word or phrase, and outputs a list of web pages that contain the word or phrase. When we do a Google search we type in the input. Pressing the Return key causes the algorithm to run, and then the output is presented on the screen.

Running such an algorithm is effortless because the computer does all the steps. But someone (actually, a group of people) designed and implemented it, and this required very specialized knowledge and skills. This chapter is an introduction to these skills. Though our treatment is elementary, the ideas presented here—if taken further—can be applied to designing quite complex and significant algorithms.

In practice, algorithms may have complex "feedback" relationships between input and output. Input might involve our clicking on a certain icon or button, and based on this choice the algorithm might prompt us to enter further information, or even upload files. Output could be as varied as an email sent to some recipient or an object produced by a 3D printer.

For simplicity we will concentrate on algorithms that simply start with input information, act on it, and produce output information at the end. To further simplify our discussion, the input and output information will be mostly numeric or alphanumeric. This is not as limiting as it may sound. Any algorithm—no matter how complex—can be decomposed into such simple "building-block algorithms."

Although all of our algorithms could be implemented on a computer, we will not express them any particular computer language. Instead we will develop a kind of *pseudocode* that has the basic features of any high-level computer language. Understanding this pseudocode makes mastering any computer language easier. Conversely, if you already know a programming language, then you may find this chapter relatively easy reading.

Our exploration begins with *variables*.

6.1 Variables and the Assignment Command

In an algorithm, a **variable** is a symbol that can be assigned various values. As in algebra, we use letters $a, b, c, \ldots, z$ as variables. If convenient, we may subscript our variables, so x_1, x_2 and x_3 are three different variables.

Though there is no harm in thinking of a variable as a name or symbol that represents a number, in programming languages a variable actually represents a location in the computer's memory that can hold different quantities (i.e., values) at different times. But it can hold only one value at any specific time. As an algorithm runs, it can assign various values to a variable at different points in time.

An algorithm is a sequence of instructions or *commands*. The command that says the variable x is to be assigned the value of 2 is expressed as

$$x := 2,$$

which we read as "*x is assigned the value* 2" or "*x gets* 2." Once this command is executed, x stands for the number 2, at least until it is assigned some other value. If a later command is

$$x := 7,$$

then x stands for the value 7. If the next command in the algorithm is

$$y := 2 \cdot x + 1,$$

then the variable y stands for the number 15. If the next command is

$$y := y + 2,$$

then after executing it y has the value $15 + 2 = 17$.

In the context of algorithms, the term *variable* has a slightly different meaning than in algebra. In an algorithm a variable represents a specific value at any point in time, and that value can change over time. But in algebra a variable is a (possibly) indefinite quantity. The difference is highlighted in the algorithm *command* $y := y + 2$, which means y gets a new value that is its previous value plus 2. By contrast, in algebra the *equation* $y = y + 2$ has no solution.

In an algorithm there is a difference between $y := 2$ and $y = 2$. In an algorithm, an expression like $y = 2$ is interpreted as an open sentence that is either true or false. Suppose an algorithm issues the command $y := 2$. Then, afterwards, the expression $y = 2$ has the value True (T), and $y = 3$ has the value False (F). Similarly, $y = y + 2$ is F, no matter the value of y.

6.2 Loops and Algorithm Notation

Programming languages employ certain kinds of *loops* that execute sequences of commands multiple times. One of the most basic kinds of loops is called a **while loop**. It is a special command to execute a sequence of commands as long as (or *while*) an open sentence $P(x)$ involving some variable x is true. A while loop has the following structure. It begins with the word **while** and ends with the word **end**, and these two words enclose a sequence of commands. The vertical bar is just a visual reminder that the commands are all grouped together within the while loop.

while $P(x)$ **do**
| Command 1
| Command 2
| ⋮
| Command n
end

When the while loop begins running, the variable x has a certain value. If $P(x)$ is true, then the while loop executes Commands 1 through n, which may change the value of x. Then, if $P(x)$ is still true the loop executes Commands 1 through n again. It continues to execute Commands 1 through n until $P(x)$ is false. At that point the loop is finished and the algorithm moves on to whatever command comes after the while loop.

The first time the while loop executes the list of commands is called the **first iteration** of the loop. The second time it executes them is called the **second iteration**, and so on.

In summary, the while loop executes the sequence of commands 1–n over and over until $P(x)$ is false. If it happens that $P(x)$ is already false when the while loop begins, then the while loop does nothing.

Let's look at some examples. These will use the command **output** x, which outputs whatever value x has when the command is executed.

Consider the while loop on the right, after the line $x := 1$. It assigns $y := 2 \cdot x$, outputs y, replaces x with $x + 1$, and continues doing this as long as $x \leq 6$. We can keep track of this with a table. After the first iteration of the loop, we have $y = 2 \cdot 1 = 2$ and $x = 1 + 1 = 2$, as shown in the table. In any successive iteration, y is twice what x was at the end of the previous iteration, and x is one more than *it* was, as reflected in the table. At the end of the 6th iteration, $x = 7$, so $x \leq 6$ is no longer true, so the loop makes no further iterations. From the table we can see that the output is the list of numbers $2, 4, 6, 8, 10, 12$

```
x := 1
while x ≤ 6 do
    y := 2 · x
    output y
    x := x + 1
end
```

iteration	1	2	3	4	5	6
x	2	3	4	5	6	7
y	2	4	6	8	10	12

Now let's tweak this example by moving the **output** command from *inside* the loop, to *after* it. This time there is no output until the while loop finishes. The table still applies, and it shows that $y = 12$ after the last iteration, so the output is the single number 12.

```
x := 1
while x ≤ 6 do
    y := 2 · x
    x := x + 1
end
output y
```

Next, consider the example on the right. It is the same as the previous example, except it has $x := x - 1$ instead of $x := x + 1$. Thus x gets smaller with each iteration, and $x \leq 6$ is *always true*, so the while loop continues forever, never stopping. This is what is called an **infinite loop**.

```
x := 1
while x ≤ 6 do
    y := 2 · x
    x := x − 1
end
output y
```

We regard an algorithm as a set of commands that completes a task in a finite number of steps. Therefore infinite loops are to be avoided. The potential for an infinite loop is seen as a mistake or flaw in an algorithm.

Now that we understand assignment commands and while loops, we can begin writing some complete algorithms. For clarity we will use a systematic notation. An algorithm will begin with a header with the word "Algorithm," followed by a brief description of what the algorithm does. Next, the input

and the output is described. Finally comes the **body** of the algorithm, a list of commands enclosed between the words **begin** and **end**. For clarity we write one command per line. We may insert comments on the right margin, preceded by a row of dots. These comments are to help a reader (and sometimes the writer!) understand how the algorithm works; they are *not* themselves commands. (If the algorithm were written in a computer language and run on a computer, the computer would ignore the comments.)

To illustrate this, here is an algorithm whose input is a positive integer n, and whose output is the first n positive even integers. If, for example, the input is 6, the output is the list $2,4,6,8,10,12$. (Clearly this is not the most impressive algorithm. It is intentionally simple because its purpose is to illustrate algorithm commands and notation.)

Algorithm 1: computes the first n positive even integers

```
Input: A positive integer n                          (Tells reader what the
Output: The first n positive even integers             input & output is.)
begin
    x := 1
    while x ≤ n do
        y := 2·x  ..... y is the xth even integer
        output y
        x := x + 1  ..... increase x by 1
    end
end
```

In addition to while loops, most programming languages feature a so-called **for loop**, whose syntax is as follows. Here i is a variable, and m and n are integers with $m \le n$.

```
for i := m to n do
    Command
    Command
        ⋮
    Command
end
```

In its first iteration the for loop sets $i := m$, and executes the list of commands between its first and last lines. In the next iteration it sets $i := m+1$ and executes the commands again. Then it sets $i := m+2$ and executes the commands, and so on, increasing i by 1 and executing the commands in each iteration. Finally, it reaches $i := n$ in the last iteration and the commands are executed a final time. None of the commands can alter i, m and n.

To illustrate this, let's rewrite Algorithm 1 with a for loop.

Algorithm 2: computes the first n positive even integers

Input: A positive integer n
Output: The first n positive even integers
begin
  for $i := 1$ **to** n **do**
    $y := 2 \cdot i$ ..... y is the ith even integer
    output y
  end
end

6.3 Logical Operators in Algorithms

There is an inseparable connection between algorithms and logic. A while loop continues to execute as long as some open sentence $P(x)$ is true. This open sentence may even involve several variables and be made up of other open sentences joined with logical operators. For example, the following loop executes the list of commands as long as $P(x) \vee \sim Q(y)$ is true.

while $P(x) \vee \sim Q(y)$ **do**
  Command
  Command
    ⋮
end

The list of commands must change the values of x or y, so $P(x) \vee \sim Q(y)$ is eventually false, or otherwise we may be stuck in an infinite loop.

Another way that algorithms can employ logic is with what is known as the **if-then** construction. Its syntax is as follows.

if $P(x)$ **then**
  Command
  Command
    ⋮
end

If $P(x)$ is true, then this executes the list of commands between the **then** and the **end**. If $P(x)$ is false it does nothing, and the algorithm continues on to whatever commands come after the closing **end**. Of course the open sentence $P(x)$ could also be a compound sentence like $P(x) \vee \sim Q(y)$, etc.

A variation on the **if-then** command is the **if-then-else** command:

if $P(x)$ **then**
  Command
  Command
    ⋮
else
  Command
    ⋮
end

If $P(x)$ is true, this executes the first set of commands, between the **then** and the **else**. And if $P(x)$ is false it executes the second set of commands, between the **else** and the **end**.

Let's use these new ideas to write an algorithm whose input is n and whose output is $n!$. Recall that if $n = 0$, then $n! = 1$ and otherwise $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots n$. Thus our algorithm should have the following structure.

if $n = 0$ **then**
  output 1 ..... because $0! = 1$
else
  Compute $y := n!$ ..... (we need to add the lines that do this)
  output y
end

To finish it, we need to add in the lines that compute $y = 1 \cdot 2 \cdot 3 \cdot 4 \cdots n$. We do this by first setting $y = 1$ and then use a for loop to multiply y by 1, then by 2, then by 3, and so on, up to a final multiplication by n.

Algorithm 3: computes $n!$

Input: A non-negative integer n
Output: $n!$
begin
  if $n = 0$ **then**
    output 1 ..... because $0! = 1$
  else
    $y := 1$
    for $i := 1$ **to** n **do**
      $y := y \cdot i$
    end
    output y ..... because now $y = n!$
  end
end

Lists often occur in algorithms. A list typically has multiple entries, so when stored in a computer's memory it's not stored in single memory location, but rather multiple locations. A list such as $X = (2,4,7,4,3)$, of length five, might be stored in six successive locations, with the first one (called X) containing the length of X:

5	2	4	7	4	3
X	x_1	x_2	x_3	x_4	x_5

The memory location X contains the number 5, which indicates that the next five locations store the five entries of the list X. We denote by x_1 the location immediately following X, and the one after that is x_2, and so on.

If an algorithm issues the command $X := (2,4,7,4,3)$, it has created a list with first entry $x_1 = 2$, second entry $x_2 = 4$, and so on. If a later command is (say) $x_3 := 1$, then we have $X = (2,4,1,4,3)$. If we then issued the for loop

for $i := 2$ **to** 5 **do**
  $x_i := 0$
end

the list becomes $X = (4,0,0,0,0)$, etc.

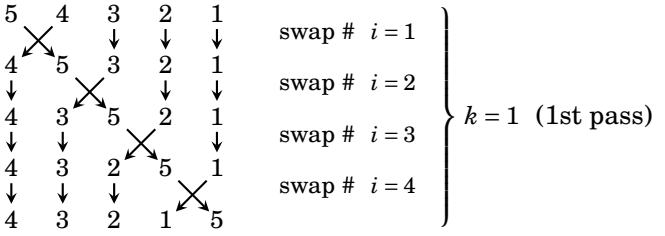
We use uppercase letters to denote lists, while their entries are denoted by a same letter in lowercase, subscripted. Thus if $A = (7,6,5,4,3,2,1)$, then $a_1 = 7$, $a_2 = 6$, etc. The command $X := A$ results in $X = (7,6,5,4,3,2,1)$.

The next algorithm illustrates these ideas. It finds the largest entry of a list. We will deviate from our tendency to use letters to stand for variables, and use the word *biggest* as a variable. The algorithm starts by setting *biggest* equal to the first list entry. Then it traverses the list, replacing *biggest* with any larger entry it finds.

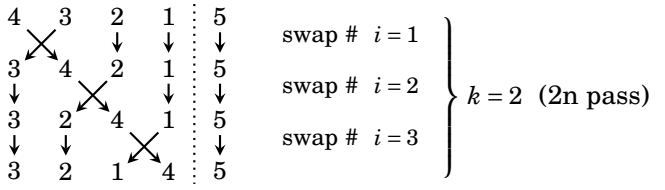
Algorithm 4: finds the largest entry of a list

Input: A list $X = (x_1, x_2, \ldots, x_n)$
Output: The largest entry in the list
begin
  $biggest := x_1$ ..... this is the largest value found so far
  for $i := 1$ **to** n **do**
    if $biggest < x_i$ **then**
      $biggest := x_i$ ..... this is the largest value found so far
    end
  end
  output $biggest$
end

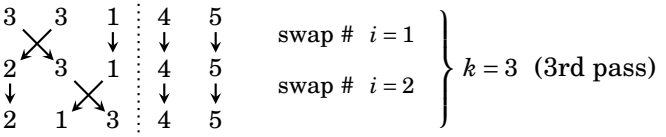
Next we create an algorithm that sorts a list into numerical order. For example, if the input is $X = (4,5,1,2,1,3)$, the output will be $X = (1,1,2,3,4,5)$. To illustrate the idea, take a very disordered list $X = (5,4,3,2,1)$. Starting at the first entry, it and the second entry are out of order, so swap them to get a new list $X = (4,5,3,2,1)$, shown on the second row below. Then move to the second entry of this new X. It and the third entry are out of order, so swap them. Now $X = (4,3,5,2,1)$ as on the third row below. Continue, in this pattern, moving left to right. For this particular list, four swaps occur.



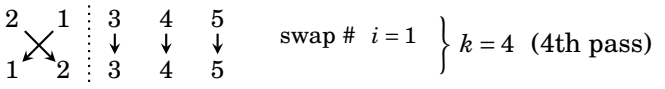
Now the last entry is in correct position, but those to its left are not. Make a second pass through the list, swapping any out of order pairs. But we can stop just before reaching the last entry, as it is placed correctly:



Now the last two entries are in their correct places. Make another pass through the list, this time stopping two positions from the left:



Now the last *three* entries are correct. We need only swap the first two.



This final list is in numeric order. Note that in this example the input list $X = (5,4,3,2,1)$ was totally out of order, and we had two swap every pair we encountered. In general, if a pair happens not to be out of order, we simply don't swap it. Our nest algorithm implements this plan.

In sorting the example list of length $n = 5$ on the previous page, we had to make $n-1$ passes through the list, numbered $k = 1,2,3,\ldots,n-1$. In the kth pass, we compared and swapped $i = n-k$ consecutive pairs of list entries (one less swap each time time k increases). Our algorithm carries out this pattern with a for loop letting k run from 1 to $n-1$. Inside this loop is another for loop that lets i run from 1 to $n-k$, and on each iteration comparing x_i to x_{i+1} and swapping if the first is larger than the second.

Algorithm 5: (Bubble Sort) sorts a list

Input: A list $X = (x_1, x_2, \ldots, x_n)$ of numbers
Output: The list sorted into numeric order
begin
  for $k := 1$ **to** $n-1$ **do**
    for $i := 1$ **to** $n-k$ **do**
      if $x_i > x_{i+1}$ **then**
        $temp := x_i$ ..... temporarily holds value of x_i
        $x_i := x_{i+1}$
        $x_{i+1} := temp$ .....now x_i and x_{i+1} are swapped
      end
    end
  end
  output X ..... now X is sorted
end

Computer scientists call Algorithm 5 the **bubble sort** algorithm, because smaller numbers "bubble up" to the front of the list. It is not the most efficient sorting algorithm (In Chapter 20 we'll see one that takes far fewer steps), but it gets the job done.

Our bubble sort algorithm has a for loop inside of another for loop. In programming, loops inside of loops are said to be **nested**. Nested loops are very common in the design of algorithms.

For full disclosure, Algorithm 5 has a minor flaw. You may have noticed it. What if the input list had length $n = 1$, like $X = (3)$? Then the first for loop would try to execute "**for** $k := 1$ to 0 **do**." This makes no sense, or could lead to an infinite loop. The same problem happens X is the empty list. It would be easy to insert an if-else statement to handle this possibility. In the interest of simplicity (and pedagogy) we did not do this. The purpose of our Algorithm 5 is really to illustrate the idea of bubble sort, and not to sort any real-life lists. But professional programmers must be absolutely certain that their algorithms are robust enough to handle any input.

Exercises for Sections 6.1, 6.2 and 6.3

1. The **Fibonacci sequence** is the sequence $1,1,2,3,5,8,13,21,34,55,\ldots$ whose first two terms are 1 and 1, and thereafter any term is the sum of the previous two terms. The numbers in this sequence are called **Fibonacci numbers**. Write an algorithm whose input is an integer n and whose output is the first n Fibonacci numbers.
2. A **geometric sequence** with ratio r is a sequence of numbers for which any term is r times the previous term. For example, $5,10,20,40,80,160,\ldots$ is a geometric sequence with ratio 2. Write an algorithm whose input is three numbers $a,r \in \mathbb{R}$, and $n \in \mathbb{N}$, and whose output is the first n terms of the geometric sequence with first term a and ratio r.
3. Write an algorithm whose input is two integers n and k, and whose output is $\binom{n}{k}$.
4. Write an algorithm whose input is a list of numbers $(x_1,x_2,\ldots,x_n)$, and whose output is the smallest number in the list.
5. Write an algorithm whose input is a list of numbers $(x_1,x_2,\ldots,x_n)$, and whose output is the word "YES" if the list has any repeated entries, and "NO" otherwise.
6. Write an algorithm whose input is two integers n,k and whose output is $P(n,k)$ (as defined in Fact 4.4 on page 94).
7. Write an algorithm whose input is two positive integers n,k, and whose output is the number of non-negative integer solutions of the equation $x_1+x_2+x+x_3+\cdots+x_k=n$. (See Section 4.9.)
8. Write an algorithm whose input is a list $X=(x_1,x_2,\ldots,x_n)$ and whose output is the word "YES" if $x_1 \le x_2 \le \cdots \le x_n$, or "NO" otherwise.
9. What does the following algorithm do?

Algorithm

Input: A list of numbers $(x_1,x_2,x_3\ldots,x_n)$
Output: ?
begin
  $x := 0$
  for $i = 1$ **to** n **do**
    $x := x + x_i$
  end
  output $\frac{x}{n}$
end

10. As noted at the bottom of page 174, our Algorithm 5 does not work on lists of length 1 or 0. Modify it so that it does.
11. Write an algorithm whose input is an integer n, and whose output is the nth row of Pascal's triangle.

6.4 The Division Algorithm

Many times in this book we will need to use the basic fact that any integer a can be divided by an integer $b > 0$, resulting in a quotient q and remainder r, for which $0 \le r < b$. In other words, given any two integers a and $b > 0$, we can find two integers q and r for which

$$a = qb + r, \qquad \text{and} \qquad 0 \le r < b.$$

As an example, $b = 3$ goes into $a = 17$ $q = 5$ times with remainder $r = 2$. In symbols, $17 = 5 \cdot 3 + 2$, or $a = qb + r$.

We are now going to write an algorithm whose input is two integers $a \ge 0$ and $b > 0$, and whose output is the two numbers q and r, for which $a = qb + r$ and $0 \le r < b$. That is, the output is the quotient and remainder that results in dividing a by b.

To see how to proceed, notice that if $a = qb + r$, then

$$a = \underbrace{b + b + b + \cdots + b}_{q \text{ times}} + r,$$

where the remainder r is less than b. This means that we can get r by continually subtracting b from a until we get a positive number r that is smaller than b. And then q is the number of times we had to subtract b. Our algorithm does just this. It keeps subtracting b from a until it gets an answer that is smaller than b (at which point no further b's can be subtracted). It uses a variable q that simply counts how many b's have been subtracted.

Algorithm 6: The division algorithm

Input: Integers $a \ge 0$ and $b > 0$
Output: Integers q and r for which $a = qb + r$ and $0 \le r < b$
begin
  $q := 0$ ..... so far we have subtracted b from a zero times
  while $a \ge b$ **do**
    $a := a - b$ .....subtract b from a until $a \ge b$ is no longer true
    $q := q + 1$ ..... q increases by 1 each time a b is subtracted
  end
  $r := a$ .....a now equals its original value, minus q b's
  output q
  output r
end

The division algorithm is actually quite old, and its origins are unclear. It goes back at least as far as ancient Egypt and Babylonia. Obviously it was not originally something that would be implemented on a computer. It was just a set of instructions for finding a quotient and remainder.

It has survived because it is so fundamental and useful. Actually, in mathematics the term *division algorithm* is usually taken to be the statement that any two integers a and $b > 0$ have a quotient and remainder. It is this statement that will be most useful for us later in this course.

Fact 6.1 (The Division Algorithm) Given integers a and b with $b > 0$, there exist integers q and r for which $a = qb + r$ and $0 \leq r < b$.

This will be very useful for proving many theorems about numbers and mathematical structures and systems, as we will see later in the course.

Notice that Fact 6.1 does not require $a \geq 0$, as our algorithm on the previous page did. In fact, the division algorithm in general works for any value of a, positive or negative. For example, if $a = -17$ and $b = 3$, then

$$a = qb + r$$

is achieved as

$$-17 = -6 \cdot 3 + 1,$$

that is, $b = 3$ goes into $a = -17$ $q = -6$ times, with a remainder of $r = 1$. Notice that indeed $0 \leq r \leq b$. Exercise 6.10 asks us to adapt Algorithm 6 so that it works for both positive and negative values of a.

6.5 Procedures and Recursion

In writing an algorithm, we may have to reuse certain blocks of code numerous times. Imagine an algorithm that has to sort two or more lists. For each sort, we'd have to insert code for a separate bubble sort. Rewriting code like this is cumbersome, inefficient and annoying.

To overcome this problem, most programming languages allow creation of *procedures*, which are mini-algorithms that accomplish some task. In general, a procedure is like a function $f(x)$ or $g(x, y)$ that we plug values into and get a result in return.

We will first illustrate this with a concrete example, and afterwards we will define the syntax for general procedures. Here is a procedure that computes $n!$.

Procedure Fac(n)

```
begin
    if n = 0 then
        return 1 ..... because 0! = 1
    else
        y := 1
        for i := 1 to n do
            y := y · i
        end
        return y ..... now y = n!
    end
end
```

This procedure now acts as a function called `Fac`. It takes as input a number n and returns the value $y = n!$, as specified in the **return** command on the last line. For example $\texttt{Fac}(3) = 6$, $\texttt{Fac}(4) = 24$, and $\texttt{Fac}(5) = 120$. Now that we have defined it we could use it in (say) an algorithm to compute $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Algorithm 7: to compute $\binom{n}{k}$

Input: Integers n and k, with $n \geq 0$
Output: $\binom{n}{k}$
begin
  if $(k < 0) \vee (k > n)$ **then**
    output 0 ..... in this case $\binom{n}{k} = 0$
  else
    output $\frac{\texttt{Fac}(n)}{\texttt{Fac}(k) \cdot \texttt{Fac}(n-k)}$ ..... procedure `Fac` is *called* here
  end
end

If an algorithm (like the one above) uses a previously-defined procedure, we say the algorithm **calls** the procedure.

In general, a procedure named (say) `Name` has the following syntax. The first line declares the name of the procedure, followed by a list of variables that it takes as input. The body of the procedure has a list of commands, including the **return** statement, saying what value the procedure returns.

Procedure Name(list of variables)

```
begin
    command
        ⋮
    return value
end
```

Our next example is a procedure called `Largest`. Its input is a list $(x_1, x_2, \ldots x_n)$ of numbers, and it returns the largest entry. For example, `Largest`$(7,2,3,8,4) = 8$. It is just a recasting of Algorithm 4 into a procedure.

Procedure Largest(x_1, x_2, x_3, ... ,x_n)

```
begin
    biggest := x_1 ..... this is the largest value found so far
    for i := 1 to n do
        if biggest < x_i then
            biggest := x_i ..... this is the largest value found so far
        end
    end
    return biggest
end
```

To conclude the section, we explore a significant idea called *recursion*. Although this is a far-reaching idea, it will not be used extensively in the remainder of this book. But it is a fascinating topic, even mind-boggling.

We have seen that a procedure is a set of instructions for completing some task. We also know that algorithms may call procedures, and you can imagine writing a procedure that calls another procedure. But under certain circumstances it makes sense for a procedure to call *itself*. Such a procedure is called a **recursive procedure**.

Here is an example. We will call it `RFac` (for RecursiveFactorial). It is our second procedure for computing a factorial, that is, `RFac`$(n) = n!$. It uses the fact that $n! = n \cdot (n-1)!$, which is to say `RFac`$(n) = n \cdot$ `RFac`$(n-1)$.

Procedure RFac(n)

```
begin
    if n = 0 then
        return 1 ..... because 0! = 1
    else
        return n · RFac(n − 1) ..... because n! = n · (n − 1)!
    end
end
```

To understand how it works, consider what happens when we run, say, `RFac`(5). Because $5 \neq 0$, the procedure's code says it needs to return $5 \cdot$ `RFac`(4). But before doing *this,* it needs to run `RFac`(4). But `RFac`(4) needs to return $4 \cdot$ `RFac`(3), and `RFac`(3) needs to run `RFac`(2), and so on.

Figure 6.1 helps keep track of this. Each call to `RFac` is indicated by a shaded rectangle. The rectangles are nested, one within another, reflecting the pattern in which calls to `RFac` occur within other calls to `RFac`.

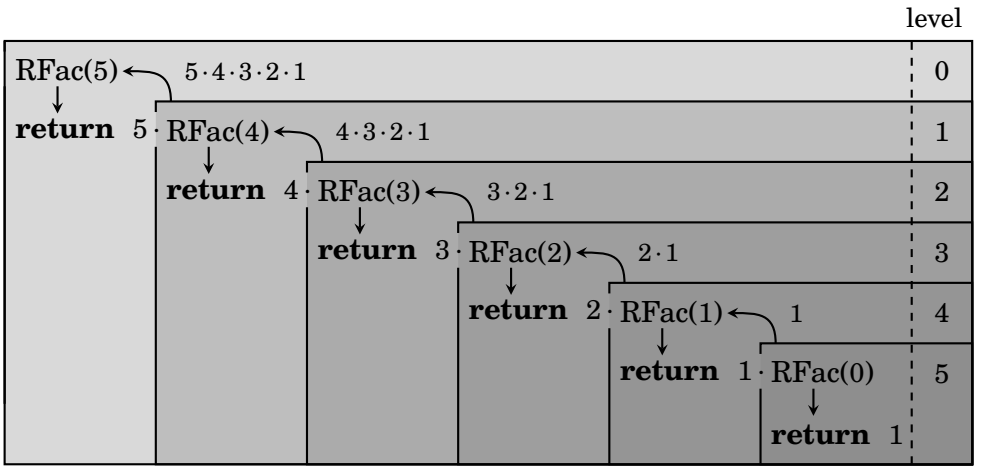


Figure 6.1. Here's what happens when we run `RFac`(5). Note that `RFac`(5) needs to return $5 \cdot$`RFac`(4). But before doing this it has to run `RFac`(4) and wait for the result. In turn, `RFac`(4) needs to return $4 \cdot$`RFac`(3), so it has to run `RFac`(3) and wait for the result. Then `RFac`(3) needs to return $3 \cdot$`RFac`(2), so it has to run `RFac`(2) and wait for the result. Next `RFac`(2) needs to return $2 \cdot$`RFac`(1), so it has to run `RFac`(1) and wait for the result. Then `RFac`(1) needs to return $1 \cdot$`RFac`(0). Here the pattern stops, as `RFac`(0) simply returns 1 (according to the procedure's code). At this point, none of the calls `RFac`(5), `RFac`(4), `RFac`(3), `RFac`(2), and `RFac`(1) is finished, because each is waiting for the next one to finish. Now `RFac`(1) returns $1 \cdot$`RFac`$(0) = 1 \cdot 1 = 1$ to `RFac`(2), which is waiting for that value. Next `RFac`(2) returns $2 \cdot$`RFac`$(1) = 2 \cdot 1$ to `RFac`(3), and `RFac`(3) returns $3 \cdot$`RFac`$(2) = 3 \cdot 2 \cdot 1$ to `RFac`(4). Finally, `RFac`(4) returns $4 \cdot$`RFac`$(3) = 4 \cdot 3 \cdot 2 \cdot 1$ to `RFac`(5). At last `RFac`(5) returns to correct vlaue of $5 \cdot$`RFac`$(4) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$.

As noted above, a procedure that calls itself is said to be a **recursive procedure**, and the situation in which a procedure calls itself (i.e., runs a copy of itself) is called **recursion**.

Some mental energy may be necessary in order to fully grasp recursion, but practice and experience will bring you to the point that you can design programs that use it. We will see recursion in several other places in this text. Section 14.2 will introduce a method of *proving* that recursion really works. In Section 20.4 designs a recursive sorting algorithm that is much quicker and more efficient than bubble sort.

Exercises for Sections 6.4 and 6.5

1. Write a procedure whose input is a list of numbers $X = (x_1, x_2, \ldots, x_n)$, and whose output is the list in reverse order.

2. Write a procedure whose input is two positive numbers n and k, and whose output is $P(n,k)$ (as defined in Fact 4.4 on page 94).

3. Write a procedure whose input is a list of numbers $X = (x_1, x_2, \ldots, x_n)$, and whose output is "YES" if X is in numeric order (i.e., $x_1 \le x_2 \le \cdots \le x_n$), and "NO" otherwise.

4. Write a procedure whose input is a list of numbers $X = (x_1, x_2, \ldots, x_n)$, and whose output is the number of entries that are negative.

5. Write a procedure whose input is a list $X = (0,0,1,0,1,\ldots,1)$ of 0's and 1's, of length n. The procedure returns the number of 1's in X.

6. Write a procedure whose input is a list of numbers $X = (x_1, x_2, \ldots, x_n)$, and whose output is the average of all the entries.

7. Write a procedure whose input is a list of numbers $X = (x_1, x_2, \ldots, x_n)$, and whose output is the product of $x_1 x_2 \cdots x_n$ of all the entries.

8. Write a procedure whose input is two lists of numbers $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$, and whose output is the merged list $Z = (x_1, y_1, x_2, y_2, x_3, y_3, \ldots, x_n, y_n)$.

9. Write a procedure whose input is two lists of numbers $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$, and whose output is the list $Z = (x_1+y_1,\ x_2+y_2,\ x_3+y_3, \ldots, x_n+y_n)$.

10. Algorithm 6 is written so that it requires $a > 0$. Rewrite it so that it works for all values of a, both positive and negative. (But still assume $b > 0$.)

11. The **Fibonacci sequence** is the sequence $1,1,2,3,5,8,13,21,34,55,\ldots$ whose first two terms are 1 and 1, and thereafter any term is the sum of the previous two terms. The numbers in this sequence are called **Fibonacci numbers**. Write a *recursive* procedure whose input is an integer n and whose output is the nth Fibonacci number.

12. A **geometric sequence** with ratio r is a sequence of numbers for which any term is r times the previous term. For example, $5,10,20,40,80,160,\ldots$ is a geometric sequence with ratio 2. Write an **recursive** procedure whose input is three numbers $a, r \in \mathbb{R}$, and $n \in \mathbb{N}$, and whose output is the nth term of the geometric sequence with first term a and ratio r.

13. An **arithmetic sequence** with difference d is a sequence of numbers for which any term is d plus the previous term. For example, $5,8,11,14,17,20,\ldots$ is a arithmetic sequence with difference 3. Write an **recursive** procedure whose input is three numbers $a, d \in \mathbb{R}$, and $n \in \mathbb{N}$, and whose output is the nth term of the arithmetic sequence whose first term is a and whose difference is d.

14. Rewrite the division algorithm (Algorithm 6 on page 176) as a recursive procedure `Div`. It should take as input two numbers a, b and return an ordered pair (q,r), where $a = qb + r$ with $0 \le r \le b$. Example: `Div`$(25,3) = (8,1)$.

6.6 Counting Steps in Algorithms

Computer scientists are attentive to algorithm *efficiency*. An algorithm should complete its task in the shortest amount of time possible, with the fewest number of steps. Of course the number of steps needed probably depends on what the input is. Thus a significant question is

How many steps does Algorithm X have to make to process input Y?

To get started, suppose an algorithm has the following piece of code, where n has been assigned an integer value in some pervious line.

```
for i := 1 to 3n do
|  Command 1
|  Command 2
end
Command 3
for j := 1 to n do
|  for k := 1 to n do
|  |  Command 4
|  end
end
```

In all, how many commands are executed? The first for loop makes $3n$ iterations, each issuing two commands, so it makes $3n \cdot 2 = 6n$ commands. Then a single command (Command 3) is executed. Next comes a nested for loop, where Command 4 is executed once for each pair (i,k) with $1 \le j,k \le n$. By the multiplication principle, there are $n \cdot n = n^2$ such pairs, so Command 4 is executed n^2 times. So in all, $6n + 1 + n^2$ commands are executed.

Now let's count the steps in this chunk of code:

```
for i := 0 to n do
|  for j := 0 to i do
|  |  for k := 0 to j do
|  |  |  Command 1
|  |  end
|  end
end
```

Command 1 is executed for each combination of i,j,k with $0 \le k \le j \le i \le n$. Each combination corresponds to a list of n stars and 3 bars $***|**|*|**\cdots*$ where k is the number of stars to the left of the first bar, j is the number of stars to the left of the second bar, and i is the number of stars to the left of the third bar. Such a list has length $n+3$, and we can make it by choosing 3 out of $n+3$ spots for the bars and filling the rest with stars.

There are $\binom{n+3}{3}$ such lists, so the number of times Command 1 is executed is $\binom{n+3}{3} = \frac{n(n-1)(n-2)}{3!} = \frac{n^3-3n^2+2n}{6} = \frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n$.

We finish the chapter by comparing two different algorithms that do the same task, namely search through a sorted list of numbers to determine if a particular number appears. We will see that the second (somewhat more complex) algorithm is vastly more efficient in terms of commands executed.

Each algorithm takes as input a number z and a list $X = \{x_1, x_2, \ldots, x_n\}$ of numbers in numeric order, that is, $x_1 \le x_2 \le \cdots \le x_n$. The output is the word "YES" if z equals some list entry; otherwise it returns the word "NO."

The first algorithm, called **sequential search**, simply traverses the list from left to right, stopping either when it finds $z = x_k$, or when it goes past the end of the list without ever finding such an x_k. A variable $found$ equals either the word "'YES" or the word "NO." The algorithm starts by assigning $found :=$ NO, and changes it to "YES" only when and if it finds a k for which $z = x_k$. It has a while loop that continues running as long as $found :=$ NO (no match found yet) and $k \le n$ (it hasn't run past the end of the list).

Algorithm 8: sequential search

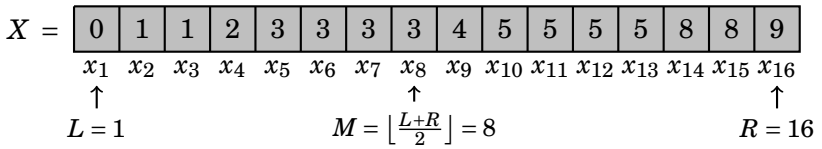
```
Input: A number z and a sorted list X = (x_1, x_2, ..., x_n) of numbers
Output: "YES" if z appears in X; otherwise "NO"
begin
    found := NO ..... means z not yet found in X
    k := 0 ..... k is subscript for list entries x_k
    while (found = NO) ∧ (k ≤ n) do
        k := k + 1 ..... go to next list entry
        if z = x_k then
            found := YES ..... the number z appears in X
        end
    end
    output found
end
```

Two comments. First, we could opt to also output k at the end of the algorithm, to tell which which list entry x_k equals z in the event of YES. Second, the sequential search algorithm also works just as well when X is not in numeric order. (But this will not be the case with our next algorithm.)

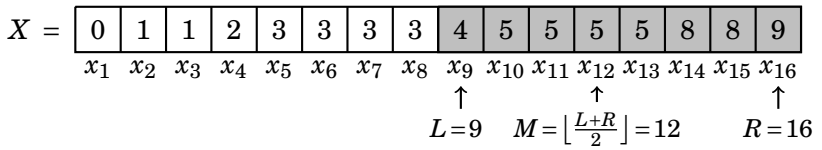
Counting steps, Algorithm 8 has two commands prior to the while loop. Then the while loop does at most n iterations, each with two commands. So it searches a list of length n in at most $2 + 2n$ steps. This is a worst-case scenario, in which z is not found, or it is found at the very end of the list. At the other extreme, if $x_1 = z$, then the algorithm stops after 4 steps.

Next we design an algorithm that takes a different approach to searching a list. Unlike sequential search, which examines every list entry, this new method ignores almost all entries but still returns the correct result.

To illustrate the idea, suppose we need to decide if $z = 4$ is in the list $X = (0,1,1,2,3,3,3,3,4,5,5,5,5,8,8,9)$. If z is in the list, it is in the shaded area between the left-most position $L = 1$ and the right-most position $R = 16$.

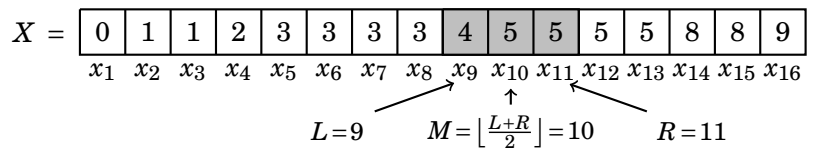


Jump to a middle position $M = \lfloor \frac{L+R}{2} \rfloor = 8$, the average of L and R, rounded down (if necessary) to an integer. The number $z = 4$ we are searching for is greater than $x_M = 3$, so it is to the right of x_8, in the shaded area below.

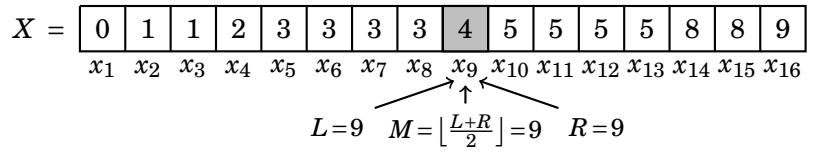


So update $L := M + 1$ and form a new middle $M := \lfloor \frac{L+R}{2} \rfloor = 12$ (shown above).

Now $x_M = 5$, and the number $z = 4$ we seek is less than x_M, so it is in the shaded area below. So update $R := M-1$. Form a new middle $M := \lfloor \frac{L+R}{2} \rfloor = 10$.



Again, $x_M = 5$, and the number $z = 4$ we seek is less than x_M, so it is in the shaded area below. Update $R := M - 1$ and form a new middle $M := \lfloor \frac{L+R}{2} \rfloor = 9$.



Now $L = R$, and we have zeroed in at $x_M = 4$, the number sought, and having ignored most entries of the list.

This new search strategy is called **binary search**. Binary search works by continually maintaining two list positions L (left) and R (right) that the searched-for entry z must be between. In each iteration, a middle M is computed. If $x_M = z$, we have found z. If $x_M < z$, then z is to the right of M, so $M+1$ becomes the new L, If $x_M > z$, then z is to the left of M, so $M-1$ becomes the new R. In this way, L and R get closer and closer to each other, trapping z between them (if indeed X contains z). If z is not in X, then eventually $L = R$. At this point the algorithm terminates and reports that z is not in X.

Algorithm 9: binary search

Input: A number z, and a sorted list $X = (x_1, x_2, \ldots, x_n)$ of numbers
Output: "YES" if z appears in X; otherwise "NO"
begin
  $found$:= NO ..... this means z has not yet been found in X
  $L := 1$ ..... left end of search area is x_1
  $R := n$ ..... right end of search area is x_n
  while $(found = \text{NO}) \wedge (L < R)$ **do**
    $M := \left\lfloor \frac{L+R}{2} \right\rfloor$ ..... M is middle of search area
    if $z = x_M$ **then**
      $found$:= YES ..... the number z appears in X
    else
      if $z < x_M$ **then**
        $R := M - 1$ ..... if z is in X, it's between x_L and x_M
      else
        $L := M + 1$ ..... if z is in X, it's between x_M and x_R
      end
    end
  end
  output $found$
end

Let's analyze the number of steps needed perform a binary search on a list of length n. Algorithm 9 starts with 3 commands, initializing $found$, L and R. Then comes the while loop, which iterates until $found = \text{YES}$ or $L = R$. How many iterations could this be? Before the first iteration, the distance between L and R is $n-1$. At each iteration, the distance between L and R is at least halved.

Thus, after the first iteration the distance between L and R is less than $\frac{n}{2}$. After the second iteration the distance between them is less than $\frac{1}{2} \cdot \frac{n}{2} = \frac{n}{2^2}$. After the third iteration the distance between them is less than $\frac{1}{2} \cdot \frac{n}{2^2} = \frac{n}{2^3}$. Thus, after i iterations, the distance between L and R is less than $\frac{n}{2^i}$.

So in the worse case, the while loop keeps running, for i iterations, until

$$\frac{n}{2^i} \leq 1 < \frac{n}{2^{i-1}},$$

which is the smallest i for which we can be confident that the distance between R and L is less than 1 (and hence 0). Multiplying this by 2^i yields

$$n \leq 2^i < 2n.$$

We can isolate the number of iterations i by taking $\log_2$, and using various logarithm properties. [1]

$$\begin{array}{rcccl} \log_2(n) & \leq & \log_2(2^i) & < & \log_2(2n) \\ \log_2(n) & \leq & i & < & \log_2(2) + \log_2(n) \\ \log_2(n) & \leq & i & < & 1 + \log_2(n). \end{array}$$

So the number of iterations i is an integer that is between $\log_2(n)$ and $1 + \log_2(n)$, which means $i = \lceil \log_2(n) \rceil$. (Generally $\log_2(n)$ is not an integer, unless $n = 2^k$ is an integer power of 2, in which case $\log_2(n) = \log_2(2^k) = k$.)

In summary, the binary search algorithm (Algorithm 9) issues 3 commands, followed by a while loop that makes at most $\lceil \log_2(n) \rceil$ iterations. Each iteration executes 2 commands: the assignment of $M = \lfloor \frac{L+R}{2} \rfloor$, followed by an if-else statement. Thus the binary search algorithm does a total of at most $3 + 2\lceil \log_2(n) \rceil$ steps to search a list of length n.

By contrast, we saw that sequential search (Algorithm 8) needs at most $2 + 2n$ steps to search a list of length n. Figure 6.2 compares the graphs of $y = 2 + 2n$ with $y = 3 + 2\log_2(n)$, showing that in general binary search involves far fewer steps than sequential search. This is especially pronounced for long lists. For example, if a list X has length $n = 2^{15} = 32768$, a sequential search could take as many as $2 + 2 \cdot 32768 = 65538$ steps, but a binary search is guaranteed to finish in no more than $3 + 2\log_2(32768) = 3 + 2 \cdot 15 = 33$ steps.

This case study illustrates a very important point. An algorithm that cannot finish quickly is of limited use, at best. In our technological world, it is often not acceptable to have to wait seconds, minutes, or hours for an

[1] If your logarithm skills are rusty, we will review logarithms in Chapter 19. They will not be used in a substantial way until Chapter 20.

algorithm to complete a critical task. Programmers need to compare the relative efficiencies of different algorithm designs, and to create algorithms that run quickly. The ability to do this rests on the foundation of the counting techniques developed in Chapter 4. We will take up this topic again, in Chapter 20, and push it further.

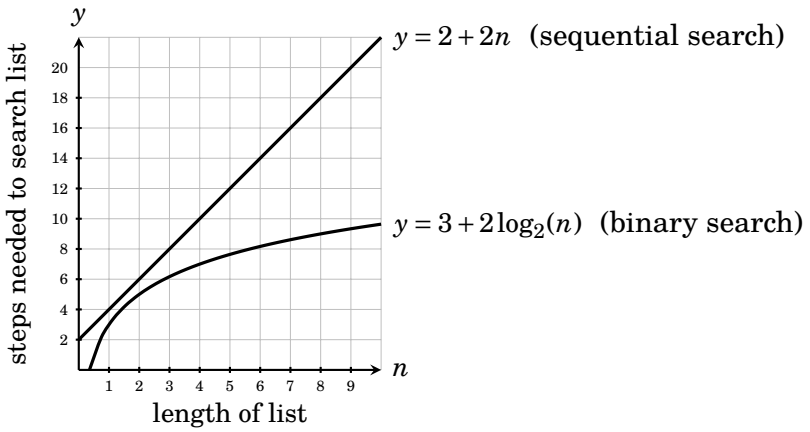


Figure 6.2. A comparison of the worst-case performance of sequential versus binary search, for lists of length n.

Exercises for Section 6.6

1. Suppose n is a positive integer. In the following piece of code, how many times is Command executed? The answer will depend on the value of n.

```
for i := 0 to n do
    for j := 0 to i do
        for k := 0 to j do
            for ℓ := 0 to k do
                Command
            end
        end
    end
end
```

2. Suppose n is a positive integer. In the following piece of code, how many times is Command executed? The answer will depend on the value of n.

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            for ℓ := 1 to n do
                Command
            end
        end
    end
end
```

3. How many steps does the bubble sort algorithm (Algorithm 5 on page 174) take if its input list $X = (x_1, x_2, \ldots, x_n)$ is already sorted?

4. Find a formula for the number of steps that Algorithm 1 (page 169) executes for an input of n.

5. Find a formula for the number of steps that Algorithm 2 (page 170) executes for an input of n.

6. Find a formula for the number of steps that Algorithm 3 (page 171) executes for an input of $n > 0$.

7. Find a formula for the number of steps that Algorithm 4 (page 172) executes when the input is a list of length n.

8. Find a formula for the worst-case number of steps that the bubble sort algorithm (Algorithm 5 on page 174) executes when the input is a list of length n.

9. Find a formula for the number of steps that The division algorithm (Algorithm 6 on page 176) executes when the input is two positive integers a and b. (The answer will depend on a and b.)