Chapter 8

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, results 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 (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 of 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 $\boxed{\text{Return}}$ 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 in 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 will make 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*.

8.1 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.

Often words or their abbreviations are used as variables. For example, a variable *rad* could represent the radius of a circle.

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, the memory location x holds the value 2, at least until it is assigned some other value. We can think of x as standing for the number 2. 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 y gets 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.

8.2 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 between these two words is 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 slightly by moving the **output** command from *inside* the loop, to *after* it, as shown on the right. This time there is no output until the while loop has finished. 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 just 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
```

Ideally, an algorithm is 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 $x \leq n$ do
        $y := 2 \cdot x$    ................................ $y$ is the $x$th 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 follows. Here i is a variable, and m, n are integers with $m \leq n$.

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

In its first iteration the for loop sets $i := m$, and executes the list of commands. In the next iteration it sets $i := m + 1$ and executes the commands again. Then it sets $i := m + 2$, executes the commands, and so on. Each iteration increases i by 1 and executes the commands. In the final iteration, $i := n$ 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
        output $2 \cdot i$    .............................. $y$ is the $i$th even integer
    end
end
```

Sometimes in a for loop, we want $m \geq n$, and for i to *decrease* from m to n. For this we allow the following construction, which is like the usual for loop except that the **to** is replaced by **downto**. Here i *decreases* from m to n in increments of 1.

```
for $i := m$ downto $n$ do
    Command
      ⋮
    Command
end
```

8.3 Logical Operators in Algorithms

There is an inseparable connection between algorithms and logic. A while loop continues to execute as long as some boolean expression $P(x)$ is true. This boolean expression may have involve multiple variables and logical operators. For example, the following loop executes the list of commands as long as $P(x) \vee \neg Q(y)$ is true.

```
while $P(x) \vee \neg Q(y)$ do
    Command
       ⋮
end
```

The list of commands must change the values of x or y, so $P(x) \vee \neg Q(y)$ is eventually false, or otherwise we will 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
       ⋮
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 $P(x) \vee \neg Q(y)$, etc.

A variation on the **if-then** command is the **if-then-else** command, shown below. If $P(x)$ is true, the commands between the **then** and the **else** are executed. But ir $P(x)$ is false, then the commands between the **else** and the **end** are run.

```
if $P(x)$ then
    Command
       ⋮
else
    Command
       ⋮
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 $y = 1{\cdot}2{\cdot}3{\cdot}4{\cdot}\cdots{\cdot}n$. 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
```

8.4 Lists and Sorting

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:

$$\begin{array}{|c|c|c|c|c|c|}\hline 5 & 2 & 4 & 7 & 4 & 3 \\ \hline \multicolumn{1}{c}{X} & \multicolumn{1}{c}{x_1} & \multicolumn{1}{c}{x_2} & \multicolumn{1}{c}{x_3} & \multicolumn{1}{c}{x_4} & \multicolumn{1}{c}{x_5} \end{array} \tag{8.1}$$

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.

With this convention, we can regard $X = x_0$ as a special location that holds the length of the list $(x_1, x_2, \ldots)$ that follows it. (So a list's length is always a known quantity that is immediately accessible without counting.)

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, that might be stored in a computer's memory in the format (8.1) above. 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.

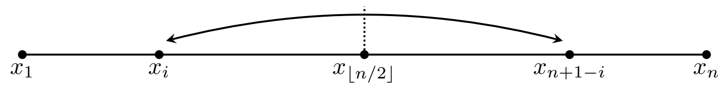
We use uppercase letters to denote lists, while their entries are denoted by a same letter in lowercase, subscripted. Thus if $A = (7, 6, 5, 4, 3, 2, 1)$, then $a_1 = 7$, $a_2 = 6$, etc. The command $X := A$ results in $X = (7, 6, 5, 4, 3, 2, 1)$.

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
```

The next example is an algorithm that reverses a list. For instance, if its input is $(1, 2, 6, 4)$ the output is $(4, 6, 2, 1)$. It uses the idea that if X is reversed, then entries x_i and x_{n+1-i} are swapped, for each $1 \leq i \leq n$, as illustrated below.

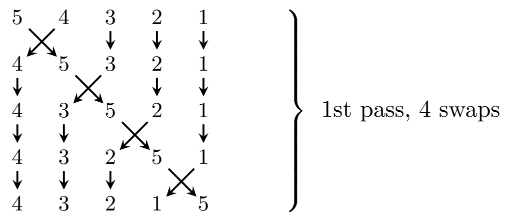


The algorithm runs through the first half of the list. For each such entry i, it stores the value of x_i in a temporary variable *temp*, overwrites x_i with x_{n+1-i}, and then assigns *temp* to x_{n+1-i}, thereby swapping the values x_i and x_{n+1-i}.

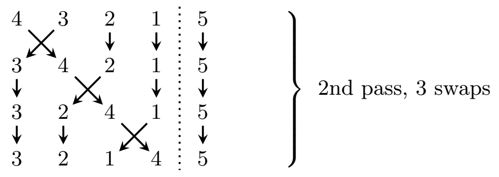
Algorithm 5: puts a list in reverse order

```
Input: A list $X = (x_1, x_2, \ldots, x_n)$ of length $n$
Output: The reverse of $X$
begin
    for $i := 1$ to $\lfloor n/2 \rfloor$ do
        $temp = x_i$
        $x_i := x_{n+1-i}$
        $x_{n+1-i} := temp$
    end
    output $X$
end
```

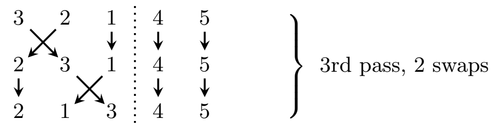
Next we create an algorithm that sorts a list into numerical order. For example, if the input is $X = (4, 5, 1, 2, 1, 3)$, the output will be $X = (1, 1, 2, 3, 4, 5)$. To illustrate the idea, take a very disordered list $X = (5, 4, 3, 2, 1)$. Starting at the first entry, it and the second entry are out of order, so swap them to get a new list $X = (4, 5, 3, 2, 1)$, shown on the second row of the diagram 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.



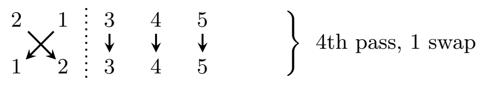
Now the last entry is in correct position, but those to its left are not. Make a second pass through the list, swapping any out-of-order pairs. But we can stop just before reaching the last entry, as it is placed correctly:



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 next algorithm implements this plan.

In sorting the example list of length $n = 5$ on the previous page, we made $n - 1$ passes through the list. In the kth pass, we compared and swapped $n-k$ consecutive pairs of list entries (one less swap for each pass).

Pass #1 compared and swapped the first $\underline{n-1}$ consecutive pairs of list entries.
Pass #2 compared and swapped the first $\underline{n-2}$ consecutive pairs of list entries.
Pass #3 compared and swapped the first $\underline{n-3}$ consecutive pairs of list entries.

$\vdots$

Pass #$(n-1)$ compared and swapped the first $\underline{\ 1\ }$ consecutive pair of list entries.

Our algorithm mimics this pattern with a for loop letting k run from $n-1$ down to 1. Inside this loop is another for loop that iterates from 1 to k, and, on the ith iteration, comparing x_i to x_{i+1} and swapping if the first is larger than the second.

Algorithm 6: (**Bubble Sort**) sorts a list

```
Input: A list $X = (x_1, x_2, \dots, x_n)$ of numbers
Output: The list sorted into numeric order
begin
    for $k := n - 1$ downto 1 do
        for $i := 1$ to $k$ 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 6 the **bubble sort** algorithm, because smaller numbers "bubble up" to the front of the list. It is not the most efficient sorting algorithm, but it works. (Chapter 21 introduces another sorting algorithm, called **merge sort**, that expends far fewer steps than bubble sort).

Algorithm 6 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.

For full disclosure, Algorithm 6 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** $k := 0$ **downto** 1 **do**." This makes no sense, or could lead to an infinite loop. The same problem happens if X is the empty list. It would be easy to insert an if-else statement to handle this, but in the interest of simplicity (and pedagogy) we did not. The purpose of our Algorithm 6 is just 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 8.1–8.4

1. Find the output.

```
$x := 1$
$y := 10$
while $x^2 < y$ do
    $y := y + x$
    $x := x + 1$
end
output $x$
output $y$
```

2. Find the output.

```
$s := 0$
$t := 0$
for $i := 1$ to $4$ do
    $s := s + t$
    $t := t + i$
end
output $s$
output $t$
```

3. Find the output.

```
$a := 0$
$b := 3$
for $i := 1$ to $8$ do
    if $a < b$ then
        $a := a + i$
    else
        $b := b + a$
    end
end
output $a$
output $b$
```

4. Find the output if the input is $X = (3, 6, 4, 9, 5, 1, 6, 2, 5, 7)$.

Algorithm

```
Input: $X = (x_1, x_2, \ldots, x_n)$
begin
    for $i := 2$ to $n$ do
        $z := x_{i-1}$
        $x_{i-1} := x_i$
        $x_i := z$
    end
    output $X$
end
```

5. Input is a list of even length. Find the output for input $X = (3, 5, 8, 4, 6, 8, 7, 4, 2, 3)$.

Algorithm

```
Input: $X = (x_1, x_2, \ldots, x_n)$
begin
    for $i := 1$ to $\frac{n}{2}$ do
        $k := 2i$
        $x_k := x_k + 1$
    end
    for $j := 1$ to $\frac{n}{2}$ do
        $k := 2j - 1$
        $x_k := x_k - 1$
    end
    output $X$
end
```

6. The following algorithm accepts a list X of numbers as input. What does the algorithm do?

Algorithm

```
Input: $X = (x_1, x_2, \ldots, x_n)$
Output: ?
begin
    $x := 0$
    for $i = 1$ to $n$ do
        $x := x + x_i$
    end
    output $\dfrac{x}{n}$
end
```

7. 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.

8. 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.

9. Write an algorithm whose input is two integers n and k, and whose output is $\binom{n}{k}$.

10. 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.

11. 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.

12. Write an algorithm whose input is two integers n, k and whose output is $P(n, k)$ (as defined in Fact 6.4 on page 124).

13. 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 6.8.)

14. Write an algorithm whose input is a list $X = (x_1, x_2, \ldots, x_n)$ and whose output is the word "YES" if $x_1 \leq x_2 \leq \cdots \leq x_n$, or "NO" otherwise.

15. As noted at the bottom of page 212, our Algorithm 6 does not work on lists of length 1 or 0. Modify it so that it does.

16. Write an algorithm whose input is a list $X = (x_1, \ldots, x_n)$, and whose output is the list X in reverse order. (For example input $(1, 3, 2, 3)$ yields output $(3, 2, 3, 1)$.)

17. Write an algorithm whose input is an integer n, and whose output is the nth row of Pascal's triangle.

8.5 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 $0 \leq r < b$. In other words, given any two integers a and $b > 0$, we can find two integers q and r for which

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

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 $a \geq 0$ and $b > 0$, and whose output is the two numbers q and r, for which $a = qb + r$ and $0 \leq r < b$. 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 non-negative 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). A variable q simply counts how many b's have been subtracted.

Algorithm 7: The division algorithm

```
Input: Integers $a \geq 0$ and $b > 0$
Output: Integers $q$ and $r$ for which $a = qb + r$ and $0 \leq r < b$
begin
    $q := 0$    ..................so far we have subtracted $b$ from $a$ zero times
    while $a \geq b$ do
        $a := a - b$    ..........subtract $b$ from $a$ until $a \geq b$ 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.

Actually, in mathematics the term *division algorithm* is usually understood 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 8.1. (The Division Algorithm) Given integers a and b with $b > 0$, there exist unique 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 8.1 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 8.12 asks us to adapt Algorithm 7 so that it works for both positive and negative values of a.

8.6 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 (or programs) that accomplish some specific 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 $n!$. Think of it as a declaration of a function $f(n) = n!$.

```
Procedure Fac($n$)
begin
    if $n = 0$ then
        return 1 ............................................ because $0! = 1$
    else
        $y := 1$
        for $i := 1$ to $n$ do
            $y := y \cdot i$
        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 `Fac`$(3) = 6$, Fac$(4) = 24$, and `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 8: 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
```

(This purpose of Algorithm 8 is only to show how a procedure might be used. It is *not* the best way to compute $\binom{n}{n}$ because the numerator and denominator of the fraction can be very large.)

If an algorithm uses a previously-defined procedure, we say the algorithm **calls** the procedure. For example, Algorithm 8 makes three calls to the procedure `Fac`.

In general, the pseudocode for 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.

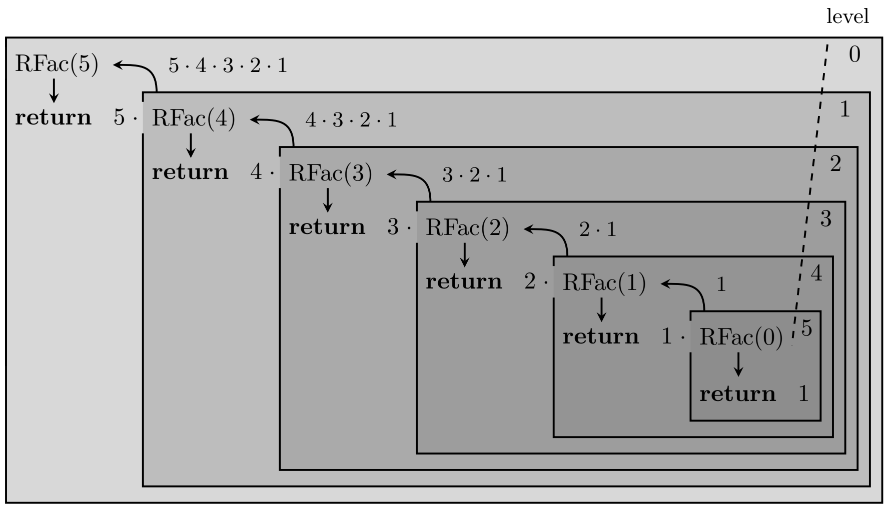
We have seen that a procedure is a set of instructions for completing some task. We also know that algorithms may call procedures, and you can imagine writing a procedure that calls another procedure. Under certain circumstances it makes sense for a procedure to call *itself.* Such a procedure is called 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 $n \cdot$ RFac($n-1$) .................... because $n! = n \cdot (n-1)!$
    end
end
```

To understand how it works, consider what happens when we run, say, RFac(5). In running RFac(5), the procedure's code says it must return the value $5 \cdot$ RFac(4). Before doing *this,* it must run RFac(4). So it runs RFac(4) and waits for the result. But then RFac(4) must return $4 \cdot$RFac(3), so *it* runs RFac(3) and waits for the result. But then RFac(3) must return $3 \cdot$RFac(2), so *it* runs RFac(2) and waits for the result. But then RFac(2) must return $2 \cdot$RFac(1), so *it* runs RFac(1) and waits for the result. Finally, RFac(1) just returns 1, which RFac(2) was waiting for.
Then RFac(2) returns $2 \cdot$ RFac(1) $= 2 \cdot 1$, which RFac(3) was waiting for.
Then RFac(3) returns $3 \cdot$ RFac(2) $= 3 \cdot 2 \cdot 1$, which RFac(4) was waiting for.
Then RFac(4) returns $4 \cdot$ RFac(3) $= 4 \cdot 3 \cdot 2 \cdot 1$, which RFac(5) was waiting for.
At last, RFac(5) can return $5 \cdot$ RFac(4) $= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$.

The diagram below is a schematic of the running of RFac(5). 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.



A procedure that calls itself is a **recursive procedure**. 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 15.5 will introduce a method of *proving* that recursion really works. Section 21.5 introduces a recursive sorting algorithm that is quicker and more efficient than bubble sort.

Exercises for Sections 8.5 and 8.6

1. Write a procedure whose input is a list of numbers $X = (x_1, x_2, \dots, x_n)$, 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 6.4 on page 124).

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

5. Write a procedure whose input is a list $X = (0, 0, 1, 0, 1, \dots, 1)$ 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 $X = (x_1, x_2, \dots, x_n)$, and whose output is the average of all the entries.

7. Write a procedure whose input is a list of numbers $X = (x_1, x_2, \dots, x_n)$, 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 a list of numbers $X = (x_1, x_2, \dots, x_n)$, and whose output is the list $(x_1, 2x_2, 3x_3, \dots, nx_n)$.

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

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

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

12. Algorithm 7 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 $b > 0$.)

13. The **Fibonacci sequence** is $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$. The first two terms are 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.

14. A **geometric sequence** with ratio r is a sequence of numbers for which any term is r times the previous term. For example, $5, 10, 20, 40, 80, 160, \dots$ 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.

15. An **arithmetic sequence** with difference d is a sequence of numbers for which any term is d plus the previous term. For example, $5, 8, 11, 14, 17, 20, \dots$ 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.

8.7 Counting Steps in Algorithms

Computer scientists are very attentive to algorithm *efficiency.* An ideal algorithm completes its task as quickly as possible, with the fewest number of steps. Thus counting steps is an important problem. 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 in order to process input Y?

This section explains how to answer such questions. But, as we will see, the answer can be highly sensitive to the input Y. Ultimately, therefore, we will phrase the question as "*What is the maximum number of steps Algorithm X might have to expend in processing the input of a given size?*" In other words, "*In the worst case, how many steps are needed?*" This is still a very valid question, for if an algorithm's worst-case performance is still favorable, then it is a good algorithm.

To get started, we will look at fragments of algorithms. Suppose an algorithm has the following piece of code, where n has been assigned an integer value in a previous 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 Command 3 executes. Next comes a nested for loop, where Command 4 executes once for each pair (i, k) with $1 \leq j, k \leq n$. By the multiplication principle, there are $n{\cdot}n = n^2$ such pairs, so Command 4 executes n^2 times. So in all, a total of $6n + 1 + n^2$ commands are executed.

Similarly, in the following three-tier for loop, a command is issued once for every triple (i, j, k) with $0 \leq i \leq n,\ 0 \leq j \leq n$ and $0 \leq k \leq n.$ There are $n+1$ possibilities for each of i, j and k so Command 1 get executed $(n+1)^3$ times.

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

Now let's count the steps in this next chunk of code, which very similar to the above example, except that j and k don't go all the way up to n when $i < n$.

```
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 $0 \leq k \leq j \leq i \leq n$. 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 $\binom{n+3}{3} = \frac{n(n-1)(n-2)}{3!} = \frac{n^3-3n^2+2n}{6} = \frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n.$

Now that we've seen some examples, let's sharpen our focus. Our goal is to determine how many steps an algorithm makes to process a given input. To attain this goal we must first specify exactly what we mean by "step." A **step** in an algorithm is one of three types of commands, or operations.

- an assignment command (possibly involving a numeric computation)
- a **return** or **output** command
- an evaluation of a boolean expression.

To be sure, by this criterion some steps are more involved than others. For example, consider the following two steps.

$$k := k + 1$$
$$y := \frac{\sqrt{x} + \log_2(x)}{x^k}$$

Certainly the second one is more involved than the first, as it entails a computation that can be broken down into several numeric operations. Still, we maintain that it is a single step. The idea is that a step is an atomic action in the running of an algorithm that could be preformed in a fixed unit of time (like a nanosecond).

Let's now count the steps made by the procedure `Largest` (page 217). It returns the largest value in a list $(x_1, \ldots, x_n)$ of n numbers. It is repeated below.

Procedure Largest($x_1,\ x_2,\ x_3,\ \ldots\ ,x_n$)

```
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
```

In the first line, an assignment *biggest* $:= x_1$ is made. So far, that's one step. The last line returns *biggest*. That's another step. Between them is the for-loop, which does n iterations. Each iteration issues an if-statement. This statement makes one boolean evaluation, checking whether (*biggest* $< x_i$) is true. That counts as a step. If (*biggest* $< x_i$) happens to be true, then *another* step (*biggest* $:= x_i$) is issued. Thus the total number of steps executed by the procedure is completely dependent on if (and how many times) the expression (*biggest* $< x_i$) is true.

For instance, if the input list is $(x_1, x_2, \ldots, x_n) = (1, 2, 3, \ldots, n)$, then (*biggest* $< x_i$) is true at each iteration of the loop. Consequently, each of the n iterations does two steps, namely the evaluation (*biggest* $< x_i$) and the assignment *biggest* $:= x_i$. Therefore the total number of steps is $2 + 2n$. This is a worst-case scenario among all input lists of length n, because each loop iteration involves the maximum number (two) of steps.

At the other extreme, consider input list $(x_1, x_2, \ldots, x_n) = (n, n{-}1, n{-}2, \ldots, 1)$ (the reverse of the list in the previous paragraph). This time (*biggest* $< x_i$) is *false* at each iteration of the loop, so the assignment *biggest* $:= x_i$ is never made. The total number of steps is only $2 + n$.

In summary, when processing a list of length n, the procedure `Largest` does at least $2 + n$ steps, and at most $2 + 2n$ steps. The function $f(n) = 2 + 2n$ measures the worst-case performance of the procedure, in the sense that an input of size n can always be processed in $f(n)$ *or fewer* steps. This suggests a definition.

Definition 8.1. An algorithm has **performance no worse than** $f(n)$, provided that when processing an input of size n, it never makes more than $f(n)$ steps.

The next section further illustrates Definition 8.1. By finding the worst-case performances of two algorithms, we can categorically say which one is better.

Exercises for Section 8.7

1. Count how many times Command is executed.

```
for $i := 1$ to $60$ do
    for $j := 1$ to $i$ do
        Command
    end
end
```

2. Count how many times Command is executed.

```
for $i := 1$ to $60$ do
    for $j := 1$ to $i$ do
        Command
    end
end
```

3. Let n be a positive integer. How many times is Command executed? (The answer depends on n.)

```
for $i := 0$ to $n$ do
    for $j := 0$ to $i$ do
        for $k := 0$ to $j$ do
            for $\ell := 0$ to $k$ do
                Command
            end
        end
    end
end
```

4. Suppose n is a positive integer. How many times is Command executed? (The answer depends on n.)

```
for $i := 1$ to $n$ do
    for $j := 1$ to $n$ do
        for $k := 1$ to $n$ do
            for $\ell := 1$ to $n$ do
                Command
            end
        end
    end
end
```

5. Count how many times Command is executed.

```
for $i := 1$ to $2017$ do
    if $i$ is even then
        Command
    else
        Command
        Command
    end
end
```

6. Count how many times Command is executed.

```
for $i := 0$ to $4$ do
    for $j := 0$ to $40$ do
        for $k := 0$ to $400$ do
            Command
        end
    end
end
```

7. How many steps does the bubble sort algorithm (Algorithm 6 on page 212) take if its input list $X = (x_1, x_2, \ldots, x_n)$ is already sorted?

8. Find the number of steps that Algorithm 1 (page 207) executes for an input of n.

9. Find the number of steps that Algorithm 2 (page 207) executes for an input of n.

10. Find the number of steps that Algorithm 3 (page 209) executes for an input of $n > 0$.

11. Find number of steps that Algorithm 4 (page 210) executes when the input is a list of length n.

12. Find a formula for the worst-case number of steps that the bubble sort algorithm (Algorithm 6 on page 212) executes when the input is a list of length n.

13. Find the number of steps that the division algorithm (Algorithm 7 on page 215) executes for an input of two integers a and b. (The answer depends on a and b.)

8.8 Case Study: Sequential Search versus Binary Search

We finish the chapter by comparing two different algorithms that do the same task, namely determine if a certain number appears on a sorted list. We will see that the second (more complex) one is vastly more efficient in terms of steps executed.

Each algorithm takes as input a number z and a list $X = (x_1, x_2, \dots, x_n)$ of numbers in numeric order, that is, $x_1 \leq x_2 \leq \cdots \leq x_n$. The output is the word "YES" if z equals some list entry; otherwise the output is 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 reaches list's end. 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 $k \leq n$.

Algorithm 9: sequential search

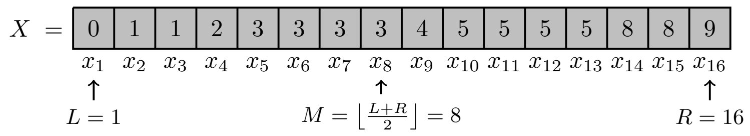
```
Input: A number $z$ and a sorted list $X = (x_1, x_2, \dots, x_n)$ of numbers
Output: "YES" if $z$ appears in $X$; otherwise "NO"
begin
    found := NO ................................ means $z$ not yet found in $X$
    $k := 0$ .................................. $k$ is subscript for list entries $x_k$
    while ( found = NO ) $\wedge$ $(k < n)$ 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
```

Notice that sequential search works just as well when X is not in numeric order. (But this will not be the case with our next algorithm.)

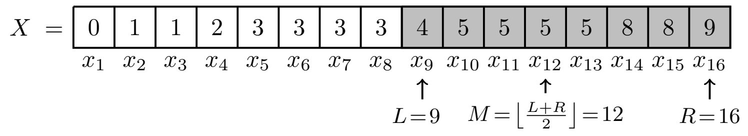
Counting steps, Algorithm 9 has two steps before the while-loop, and one after it. The loop does at most n iterations, each involving at most four steps (two boolean evaluations and two assignments). So it finishes in at most $f(n) = 3 + 4n$ steps. This is a worst-case scenario, in which z is not found, or it is found at the end of the list. (At the other extreme, if $x_1 = z$, then the algorithm stops after 7 steps.)

Now let's develop our second list searching algorithm. Unlike sequential search, which examines every list entry, this new method ignores almost all entries but still returns the correct result. It is akin to looking up a word in an old-fashioned dictionary. Opening the book to the middle page, you see that that word you seek appears (say) *after* this page. You then ignore the first half of the book and repeat the process on the second half, in essence halving the book at each step until you zero in on the word.

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 $R = 16$.

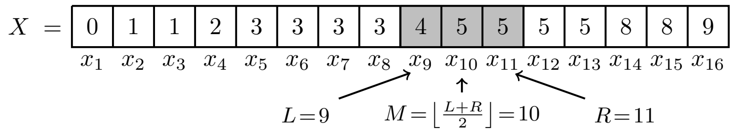

$X =$
0 1 1 2 3 3 3 3 4 5 5 5 5 8 8 9
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16}
$L = 1$
$M = \lfloor \frac{L+R}{2} \rfloor = 8$
$R = 16$

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.

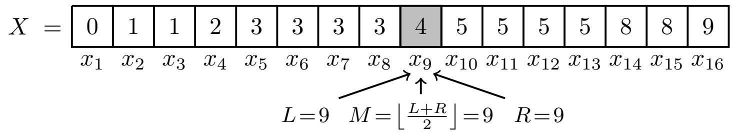

$X =$
0 1 1 2 3 3 3 3 4 5 5 5 5 8 8 9
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16}
$L = 9$
$M = \lfloor \frac{L+R}{2} \rfloor = 12$
$R = 16$

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 $M := \lfloor \frac{L+R}{2} \rfloor = 10$.


$X =$
0 1 1 2 3 3 3 3 4 5 5 5 5 8 8 9
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16}
$L = 9$
$M = \lfloor \frac{L+R}{2} \rfloor = 10$
$R = 11$

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 $M := \lfloor \frac{L+R}{2} \rfloor = 9$.


$X =$
0 1 1 2 3 3 3 3 4 5 5 5 5 8 8 9
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16}
$L = 9$
$M = \lfloor \frac{L+R}{2} \rfloor = 9$
$R = 9$

Now $L = R$, and we have zeroed in at $x_M = 4$, the number sought.

This new search strategy is called **binary search**. Binary search continually maintains 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 $x_M = z$, then z is found. 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 10: 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 count the steps needed perform a binary search on a list. Algorithm 10 starts with 3 commands, initializing *found*, L and R. It closes with one output statement. Between these is the while loop, which iterates until *found* = YES or $L = R$. How many iterations is this? 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 $\frac{1}{2} \cdot \frac{n}{2} = n/2^2$. After the third iteration the distance between them is less than $\frac{1}{2} \cdot \frac{n}{2^2} = n/2^3$. Thus, after k iterations, the distance between L and R is less than $n/2^k$.

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

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

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

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

We can isolate the number of iterations k by taking $\log_2$, and applying logarithm properties. (If your logarithm skills are rusty, Chapter 20 is a review. Logarithms will not be used in a substantial way until Chapter 21.)

$$\begin{aligned} n &\leq 2^k < 2n \\ \log_2(n) &\leq \log_2(2^k) < \log_2(2n) \\ \log_2(n) &\leq k < \log_2(2) + \log_2(n) \\ \log_2(n) &\leq k < 1 + \log_2(n). \end{aligned}$$

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

In summary, binary search (Algorithm 10) issues four commands outside the while-loop, and the loop that makes at most $\lceil \log_2(n) \rceil$ iterations. Each iteration executes at most five commands (check this). Thus the binary search algorithm does a total of at most $g(n) = 4 + 5\lceil \log_2(n) \rceil$ steps to search a list of length n.

By contrast, recall that sequential search (Algorithm 9) needs at most $f(n) = 3 + 4n$ steps to search a list of length n. Figure 8.1 compares the graphs of $f(n) = 3 + 4n$ with $g(n) = 4 + 5\log_2(n)$, 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 has length $n = 2^{15} = 32768$, a sequential search could take as many as $3 + 4 \cdot 32768 = 131075$ steps, but binary search is guaranteed to finish in no more than $4 + 5\log_2(32768) = 4 + 5 \cdot 15 = 79$ steps.

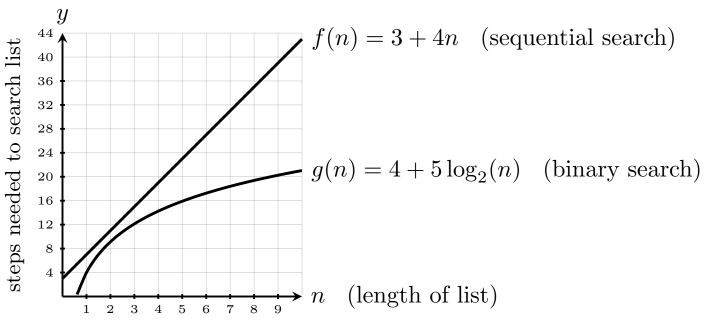


Fig. 8.1 A comparison of the worst-case performance of sequential versus binary search.

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 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 6. We will take up this topic again, in Chapter 21, and push it further.

Solutions for Chapter 8

Sections 8.1, 8.2 and 8.3

1. Find the output.

```
$x := 1$
$y := 10$
while $x^2 < y$ do
    $y := y + x$
    $x := x + 1$
end
output $x$
output $y$
```

Solution The following table tallies the values of x and y initially, and at the end of each iteration of the loop.

iteration		1	2	3
x	1	2	3	4
y	10	11	13	16

The final values (which are the output) are $x = 4$ and $y = 16$.

3. Find the output.

```
$a := 0$
$b := 3$
for $i := 1$ to 8 do
    if $a < b$ then
        $a := a + i$
    else
        $b := b + a$
    end
end
output $a$
output $b$
```

Solution The following table tallies the values of a and b initially, and at the end of each iteration (i) of the loop.

iteration (i)		1	2	3	4	5	6	7	8
a	0	1	3	3	7	7	13	13	21
b	3	3	3	6	6	13	13	26	26

The final values (which are the output) are $a = 21$ and $b = 26$.

5. The input of the following algorithm is a list X of even length. Find the output for input $X = (3, 5, 8, 4, 6, 8, 7, 4, 2, 3)$.

```
Algorithm
Input: $X = (x_1, x_2, \dots, x_n)$
begin
    for $i := 1$ to $\frac{n}{2}$ do
        $k := 2i$
        $x_k := x_k + 1$
    end
    for $j := 1$ to $\frac{n}{2}$ do
        $k := 2j - 1$
        $x_k := x_k - 1$
    end
    output $X$
end
```

Solution The first for loop adds 1 to each list entry x_k for which the index k is even. In other words, it adds 1 to the entries x_2, x_4, x_6, x_8 and x_{10}.

The second for loop subtracts 1 from each list entry x_k for which the index k is odd. In other words, it subtracts 1 from the entries x_1, x_3, x_5, x_7 and x_9.

Therefore the output is
$X = (2, 6, 7, 5, 5, 9, 6, 5, 1, 4)$.

7. Write an algorithm whose input is an integer n and whose output is the first n Fibonacci numbers.

Algorithm: to compute the first n Fibonacci numbers

```
Input: An integer $n$ for which $n \geq 2$
Output: The first $n$ Fibonacci numbers
begin
    $x := 1$ .................. $x$ is the 1st Fibonacci number
    $y := 1$ .................. $y$ is the 2nd Fibonacci number
    output $x$ ................ output 1st Fibonacci number
    output $y$ ................ output 2nd Fibonacci number
    $i := 2$ .................. $i$ is # of Fibonacci numbers outputted so far
    while $i < n$ do
        $z := x + y$ .......... $z$ is most recent Fibonacci number
        output $z$ ............ output most recent Fibonacci number
        $i := i + 1$ .......... update $i$
        $x := y$ .............. $x$ now second-most-recent Fibonacci number
        $y := z$ .............. $y$ now most recent Fibonacci number
    end
end
```

9. Write an algorithm whose input is two integers n and k, and whose output is $\binom{n}{k}$.
Solution: Recall that $\binom{n}{k} = 0$ if $n \leq 0$, or if $n > 0$ but $k < 0$ or $k > n$. Also $\binom{n}{k} = 1$ when $k = 0$ or $k = n$. Otherwise, Fact 6.5 (page 128) says

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdots(n-k+3)(n-k+2)(n-k+1)}{k!}$$
$$= \frac{(n-k+1)(n-k+2)(n-k+3)\cdots(n-2)(n-1)\,n}{1\cdot 2\cdot 3\cdots(k-2)(k-1)\,k}.$$

Our algorithm will carry out this arithmetic by first putting $y := 1$, then using a for loop to multiply y by $(n-k+1)$, then by $(n-k+2)$, then $(n-k+3)$, and so on, working its way up to multiplying by n. Then a second for loop will divide by 1, then by 2, then by 3, and so on, until finally dividing by k.

Algorithm: computes $\binom{n}{k}$

```
Input: Integers $n$ and $k$, with $n \geq 0$
Output: $\binom{n}{k}$
begin
    if $(n \leq 0) \vee \big((k < 0) \vee (k > n)\big)$ then
        output 0 .................. in this case $\binom{n}{k} = 0$
    else
        if $(k = 0) \vee (k = n)$ then
            output 1 .............. in this case $\binom{n}{k} = 1$
        else
            $y := 1$ .............. $y$ is initially 1
            for $i := n - k + 1$ to $n$ do
                $y := y \cdot i$ ...... multiply $y$ by $i$, for each $n - k + 1 \leq i \leq n$
            end
            for $i := 1$ to $k$ do
                $y := \frac{y}{i}$ ...... divide $y$ by $i$, for each $1 \leq i \leq k$
            end
            output y .............. now $y = \binom{n}{k}$
        end
    end
end
```

11. Write an algorithm whose input is a list of numbers $(x_1, x_2, \dots, x_n)$, and whose output is the word "YES" if the list has any repeated entries, and "NO" otherwise.

Solution: For each x_i up to x_{n-1} we check if it equals an x_k later on the list.

Algorithm

```
Input: A list of numbers $x_1, x_2, x_3 \dots, x_n$
Output: "YES" if the list has repetition, otherwise "NO"
begin
    $match :=$ NO
    for $i := 1$ to $n-1$ do
        for $k = i+1$ to $n$ do
            if $x_i = x_k$ then
                $match :=$ YES
            end
        end
    end
end
output $match$
```

13. Write an algorithm whose input is two positive integers n, k, and whose output is the number of non-negative integer solutions of $x_1 + x_2 + x + x_3 + \cdots + x_k = n$.

Solution: As in Section 6.8 we can model the solutions with stars-and-bars lists

$$\overbrace{***\cdots*}^{x_1} \mid \overbrace{***\cdots*}^{x_2} \mid \overbrace{***\cdots*}^{x_3} \mid \cdots \mid \overbrace{***\cdots*}^{x_k},$$

having n stars and $k-1$ bars. Such a list has length $n+k-1$, and can be made by choosing n positions for stars and filling the remaining $k-1$ with bars. Thus there are $\binom{n+k-1}{n}$ such lists, so this is also the number of solutions to the equation. Thus our algorithm must simply compute $\binom{n+k-1}{n}$. For this we can adapt the algorithm for $\binom{n}{k}$ in Exercise 9 above.

Algorithm: computes $\binom{n+k-1}{n}$

```
Input: Positive integers $n$ and $k$
Output: $\binom{n+k-1}{n}$
begin
    $y := 1$ ............................................ $y$ is initially 1
    for $i := k$ to $n+k-1$ do
        $y := y \cdot i$ ............. multiply $y$ by $i$, for each $k \le i \le n+k-1$
    end
    for $i := 1$ to $k$ do
        $y := \frac{y}{i}$ ......................... divide $y$ by $i$, for each $1 \le i \le k$
    end
    output y ........................................ now $y = \binom{n+k-1}{n}$
end
```

15. Fix BubbleSort.

(**Better Bubble Sort**) sorts any list

```
Input: A list $X = (x_1, x_2, \ldots, x_n)$ of numbers
Output: The list sorted into numeric order
begin
    if $0 \leq n \leq 1$ then
        output $X$ .................... $X$ is already sorted
    else
        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
end
```

17. Design an algorithm whose output is the nth row of Pascal's triangle.

For an input of n, the output will be the sequence $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \binom{n}{3}, \ldots, \binom{n}{n}$.

How could we do this? To begin, we need a for loop with the following structure.

```
for $k := 0$ to $n$ do
    $y := \binom{n}{k}$
    output $y$
end
```

To finish it we just need to add in the lines that compute $y := \binom{n}{k}$. For this we can reuse our code from Algorithm 11 in our solution of Exercise 9. Actually, the above for loop makes k go from 1 to n, so we don't even need the lines of Algorithm 11 that deal with the cases $n \leq 0 \vee \big((k < 0) \vee (k > n)\big)$, for which $\binom{n}{k} = 0$.

Algorithm: computes the nth row of Pascal's triangle

```
Input: Integer $n$ with $n \geq 0$
Output: $n$th row of Pascal's triangle
begin
    for $k := 0$ to $n$ do
        if $(k = 0) \vee (k = n)$ then
            output 1 .................... in this case $\binom{n}{k} = 1$
        else
            $y := 1$ .................... $y$ is initially 1
            for $i := n - k + 1$ to $n$ do
                $y := y \cdot i$
            end
            for $i := 1$ to $k$ do
                $y := y/i$
            end
            output y .................... now $y = \binom{n}{k}$
        end
    end
end
```

Sections 8.5 and 8.6

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.

```
Procedure Reverse($X$)
begin
    $Y := X$ .................$Y = (y_1, \ldots, y_n)$ is a copy of $X = (x_1, \ldots, x_n)$
    for $i := 1$ to $n$ do
        $y_i := x_{n-i+1}$ .......................... fill in $Y$ as the reverse of $X$
    end
    return $Y$
end
```

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., $x_1 \leq x_2 \leq \cdots \leq x_n$), and "NO" otherwise.

```
Procedure Check($X$)
begin
    $ordered$ := YES ......list assumed ordered until found not to be ordered
    $i := 1$
    while ($ordered$ = YES) $\wedge$ ($i < n$) do
        if $x_i > x_{i+1}$ then
            $ordered$ := NO
        end
        $i := i + 1$
    end
    return $ordered$
end
```

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.

```
Procedure Ones($X$)
begin
    $total := 0$ .......................so far total number of 1's found is zero
    for $i := 1$ to $n$ do
        if $x_i = 1$ then
            $total := total + 1$
        end
    end
    return $total$
end
```

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.

```
Procedure Prod($X$)
begin
    $product := 1$ for $i := 1$ to $n$ do
        $product := product \cdot x_i$
    end
    return $prod$
end
```

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 $Z = (x_1, x_2, x_3, \ldots, x_n, y_n, \ldots, y_3, y_2, y_1)$.

```
Procedure Glue($X, Y$)
begin
    $Z := (0, 0, 0, \ldots, 0)$ .................................. a list of length $2n$
    for $i := 1$ to $n$ do
        $z_i := x_i$
        $z_{n+1-i} := y_i$
    end
    return $Z$
end
```

11. 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 $Z = (x_1+y_1,\ x_2+y_2,\ x_3+y_3, \ldots, x_n+y_n)$.

```
Procedure Add($X, Y$)
begin
    $Z := X$ ............................................. $Z$ is a copy of $X$
    for $i := 1$ to $n$ do
        $z_i := z_i + y_i$
    end
    return $Z$
end
```

13. The **Fibonacci sequence** is the sequence $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$. Write a *recursive* procedure whose input is an integer n and whose output is the nth Fibonacci number.

```
Procedure Fib($n$)
begin
    if $(n = 1) \vee (n = 2)$ then
        return 1
    else
        return Fib$(n-1)$ + Fib$(n-2)$
    end
end
```

15. 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.

```
Procedure Arithmetic($a, d, n$)
begin
    if $n = 1$ then
        return $a$
    else
        return $d$ + Arithmetic$(n-1)$
    end
end
```

Section 8.7

1. Count how many times Command is executed.

```
for $i := 1$ to $60$ do
    for $j := 1$ to $i$ do
        Command
    end
end
```

Solution Command is issued once for each pair (i, j) with $1 \leq j \leq i \leq 60$. Such a pair can be encoded as a star-and-bar list $***\ldots*|***\ldots*|***\ldots*$ with 60 stars and two bars, where j is the number of stars before the first bar and i is the number of stars before the second bar. Given that we have $1 \leq j$, the first list entry must be a star. The remaining entries form a list with 59 stars and two bars, of length 61. The number of such lists is $\binom{61}{2} = 1830$ so that is the number of times Command is executed.

3. 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 $j := 0$ to $j$ do
            for $\ell = 0$ to $k$ do
                Command
            end
        end
    end
end
```

Solution: Command is executed for each integer combination of i, j, k and ℓ for which $0 \leq \ell \leq k \leq j \leq i \leq n$. We can model such combinations with lists of n stars and 4 bars $***|**|*|**|**\cdots***$ where ℓ is the number of stars to the left of the first bar, k is the number of stars to the left of the second bar, j is the number of stars to the left of the third bar, and i is the number of stars to the left of the fourth bar. Such a list has length $n+4$, and we can make it by choosing 4 out of $n+4$ spots for the bars and filling the rest with stars. Thus there are $\binom{n+4}{4}$ such lists so this is also the number of times Command is executed.

5. Count how many times Command is executed.

```
for $i := 1$ to $2017$ do
    if $i$ is even then
        Command
    else
        Command
        Command
    end
end
```

Solution There are $2018/2 = 1009$ odd integers between 1 and 2017, and 1008 even integers between 1 and 2017. Because Command gets issued once for every even integer and twice for every odd integer, it gets executed a total of $1009 + 2 \cdot 1008 = 3026$ times.

7. How many steps does the bubble sort algorithm (Algorithm 6 on page 212) take if its input list $X = (x_1, x_2, \ldots, x_n)$ is already sorted?

Solution: The if-statement inside the nested for-loops gets executed once for each pair (i, k) of integers with $1 \leq i \leq k \leq n-1$. We can model such pairs as lists made of $n-1$ stars and 2 bars, such that there are i stars before the first bar, and k before the second bar.

$$\overbrace{\underbrace{\underbrace{*****}_{i \text{ stars}}\,|\,******}_{k \text{ stars}}\,|\,*******}^{n-1 \text{ stars, and 2 bars}}$$

For example, $***|*|**$ corresponds to $(i, k) = (3, 4)$, whereas $***||***$ corresponds to $(i, k) = (3, 3)$. Also $*|*****|$ means $(i, k) = (1, 6)$, and $*||*****$ is $(i, k) = (1, 1)$. Note that because $1 \leq i$, the first entry of any such list is a star. Further the length of any such list is $n+1$. To make such a list we could choose 2 of the n entries after the first star for bars, then fill out the remaining entries with stars. The number of such lists is $\binom{n}{2} = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. So the if-statement gets executed $\frac{1}{2}n^2 - \frac{1}{2}n$ times (but $x_i > x_{i+1}$ is always false, so the three statements in its body do not get executed). Thus the algorithm does $\frac{1}{2}n^2 - \frac{1}{2}n$ steps.

9. Find the number of steps that Algorithm 2 (page 207) executes for an input of n.

Solution: For each i between 1 and n, it executes 2 steps, to the total number of steps is $2n$.

11. Find the number of steps (in the worst case) that Algorithm 4 (page 210) executes when the input is a list of length n.

Solution: The algorithm starts by making one assignment ($\mathit{biggest} := x_1$) and then executes an if-statement n times. Each time the if-statement executes, it does one boolean computation ($\mathit{biggest} < x_i$) and at worse one assignment ($\mathit{biggest} := x_1$). Therefore, at worst it makes $1 + 2n$ steps.

13. Find the number of steps that The division algorithm (Algorithm 7 on page 215) executes when the input is two positive integers a and b.

Solution: There are 4 stepa outside of the while loop. The while loop goes through $\left\lceil \frac{a}{b} \right\rceil$ iterations, and each iteration executes a boolean computation and two assignments. Thus the answer is $4 + 3\left\lceil \frac{a}{b} \right\rceil$.