

# Algorithms

- 3.1 Algorithms
- 3.2 The Growth of Functions
- 3.3 Complexity of Algorithms

Many problems can be solved by considering them as special cases of general problems. For instance, consider the problem of locating the largest integer in the sequence 101, 12, 144, 212, 98. This is a specific case of the problem of locating the largest integer in a sequence of integers. To solve this general problem we must give an algorithm, which specifies a sequence of steps used to solve this general problem. We will study algorithms for solving many different types of problems in this book. For example, in this chapter we will introduce algorithms for two of the most important problems in computer science, searching for an element in a list and sorting a list so its elements are in some prescribed order, such as increasing, decreasing, or alphabetic. Later in the book we will develop algorithms that find the greatest common divisor of two integers, that generate all the orderings of a finite set, that find the shortest path between nodes in a network, and for solving many other problems.

We will also introduce the notion of an algorithmic paradigm, which provides a general method for designing algorithms. In particular we will discuss brute-force algorithms, which find solutions using a straightforward approach without introducing any cleverness. We will also discuss greedy algorithms, a class of algorithms used to solve optimization problems. Proofs are important in the study of algorithms. In this chapter we illustrate this by proving that a particular greedy algorithm always finds an optimal solution.

One important consideration concerning an algorithm is its computational complexity, which measures the processing time and computer memory required by the algorithm to solve problems of a particular size. To measure the complexity of algorithms we use big- $O$  and big- $\Theta$  notation, which we develop in this chapter. We will illustrate the analysis of the complexity of algorithms in this chapter, focusing on the time an algorithm takes to solve a problem. Furthermore, we will discuss what the time complexity of an algorithm means in practical and theoretical terms.

## 3.1 Algorithms

### Introduction

There are many general classes of problems that arise in discrete mathematics. For instance: given a sequence of integers, find the largest one; given a set, list all its subsets; given a set of integers, put them in increasing order; given a network, find the shortest path between two vertices. When presented with such a problem, the first thing to do is to construct a model that translates the problem into a mathematical context. Discrete structures used in such models include sets, sequences, and functions—structures discussed in Chapter 2—as well as such other structures as permutations, relations, graphs, trees, networks, and finite state machines—concepts that will be discussed in later chapters.

Setting up the appropriate mathematical model is only part of the solution. To complete the solution, a method is needed that will solve the general problem using the model. Ideally, what is required is a procedure that follows a sequence of steps that leads to the desired answer. Such a sequence of steps is called an **algorithm**.

#### DEFINITION 1

An *algorithm* is a finite sequence of precise instructions for performing a computation or for solving a problem.

The term *algorithm* is a corruption of the name *al-Khowarizmi*, a mathematician of the ninth century, whose book on Hindu numerals is the basis of modern decimal notation. Originally, the word *algorism* was used for the rules for performing arithmetic using decimal notation. *Algorism* evolved into the word *algorithm* by the eighteenth century. With the growing interest in computing machines, the concept of an algorithm was given a more general meaning, to include all definite procedures for solving problems, not just the procedures for performing arithmetic. (We will discuss algorithms for performing arithmetic with integers in Chapter 4.)

In this book, we will discuss algorithms that solve a wide variety of problems. In this section we will use the problem of finding the largest integer in a finite sequence of integers to illustrate the concept of an algorithm and the properties algorithms have. Also, we will describe algorithms for locating a particular element in a finite set. In subsequent sections, procedures for finding the greatest common divisor of two integers, for finding the shortest path between two points in a network, for multiplying matrices, and so on, will be discussed.

**EXAMPLE 1** Describe an algorithm for finding the maximum (largest) value in a finite sequence of integers.



Even though the problem of finding the maximum element in a sequence is relatively trivial, it provides a good illustration of the concept of an algorithm. Also, there are many instances where the largest integer in a finite sequence of integers is required. For instance, a university may need to find the highest score on a competitive exam taken by thousands of students. Or a sports organization may want to identify the member with the highest rating each month. We want to develop an algorithm that can be used whenever the problem of finding the largest element in a finite sequence of integers arises.

We can specify a procedure for solving this problem in several ways. One method is simply to use the English language to describe the sequence of steps used. We now provide such a solution.

*Solution of Example 1:* We perform the following steps.

1. Set the temporary maximum equal to the first integer in the sequence. (The temporary maximum will be the largest integer examined at any stage of the procedure.)
2. Compare the next integer in the sequence to the temporary maximum, and if it is larger than the temporary maximum, set the temporary maximum equal to this integer.
3. Repeat the previous step if there are more integers in the sequence.
4. Stop when there are no integers left in the sequence. The temporary maximum at this point is the largest integer in the sequence. ▶

An algorithm can also be described using a computer language. However, when that is done, only those instructions permitted in the language can be used. This often leads to a description of the algorithm that is complicated and difficult to understand. Furthermore, because many programming languages are in common use, it would be undesirable to choose one particular language. So, instead of using a particular computer language to specify algorithms, a form of **pseudocode**, described in Appendix 3, will be used in this book. (We will also describe algorithms using the English language.) Pseudocode provides an intermediate step between



**ABU JA'FAR MOHAMMED IBN MUSA AL-KHOWARIZMI (C. 780–C. 850)** al-Khowarizmi, an astronomer and mathematician, was a member of the House of Wisdom, an academy of scientists in Baghdad. The name al-Khowarizmi means “from the town of Kowarizim,” which was then part of Persia, but is now called *Khiva* and is part of Uzbekistan. al-Khowarizmi wrote books on mathematics, astronomy, and geography. Western Europeans first learned about algebra from his works. The word *algebra* comes from al-jabr, part of the title of his book *Kitab al-jabr w'al muqabala*. This book was translated into Latin and was a widely used textbook. His book on the use of Hindu numerals describes procedures for arithmetic operations using these numerals. European authors used a Latin corruption of his name, which later evolved to the word *algorithm*, to describe the subject of arithmetic with Hindu numerals.

an English language description of an algorithm and an implementation of this algorithm in a programming language. The steps of the algorithm are specified using instructions resembling those used in programming languages. However, in pseudocode, the instructions used can include any well-defined operations or statements. A computer program can be produced in any computer language using the pseudocode description as a starting point.

The pseudocode used in this book is designed to be easily understood. It can serve as an intermediate step in the construction of programs implementing algorithms in one of a variety of different programming languages. Although this pseudocode does not follow the syntax of Java, C, C++, or any other programming language, students familiar with a modern programming language will find it easy to follow. A key difference between this pseudocode and code in a programming language is that we can use any well-defined instruction even if it would take many lines of code to implement this instruction. The details of the pseudocode used in the text are given in Appendix 3. The reader should refer to this appendix whenever the need arises.

A pseudocode description of the algorithm for finding the maximum element in a finite sequence follows.

#### ALGORITHM 1 Finding the Maximum Element in a Finite Sequence.

```

procedure  $\text{max}(a_1, a_2, \dots, a_n)$ : integers)
 $\text{max} := a_1$ 
for  $i := 2$  to  $n$ 
    if  $\text{max} < a_i$  then  $\text{max} := a_i$ 
return  $\text{max}$  { $\text{max}$  is the largest element}

```

This algorithm first assigns the initial term of the sequence,  $a_1$ , to the variable  $\text{max}$ . The “for” loop is used to successively examine terms of the sequence. If a term is greater than the current value of  $\text{max}$ , it is assigned to be the new value of  $\text{max}$ .

**PROPERTIES OF ALGORITHMS** There are several properties that algorithms generally share. They are useful to keep in mind when algorithms are described. These properties are:

- *Input.* An algorithm has input values from a specified set.
- *Output.* From each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem.
- *Definiteness.* The steps of an algorithm must be defined precisely.
- *Correctness.* An algorithm should produce the correct output values for each set of input values.
- *Finiteness.* An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
- *Effectiveness.* It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- *Generality.* The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

**EXAMPLE 2** Show that Algorithm 1 for finding the maximum element in a finite sequence of integers has all the properties listed.

*Solution:* The input to Algorithm 1 is a sequence of integers. The output is the largest integer in the sequence. Each step of the algorithm is precisely defined, because only assignments, a finite loop, and conditional statements occur. To show that the algorithm is correct, we must show that when the algorithm terminates, the value of the variable  $\text{max}$  equals the maximum

of the terms of the sequence. To see this, note that the initial value of  $max$  is the first term of the sequence; as successive terms of the sequence are examined,  $max$  is updated to the value of a term if the term exceeds the maximum of the terms previously examined. This (informal) argument shows that when all the terms have been examined,  $max$  equals the value of the largest term. (A rigorous proof of this requires techniques developed in Section 5.1.) The algorithm uses a finite number of steps, because it terminates after all the integers in the sequence have been examined. The algorithm can be carried out in a finite amount of time because each step is either a comparison or an assignment, there are a finite number of these steps, and each of these two operations takes a finite amount of time. Finally, Algorithm 1 is general, because it can be used to find the maximum of any finite sequence of integers. ◀

## Searching Algorithms

The problem of locating an element in an ordered list occurs in many contexts. For instance, a program that checks the spelling of words searches for them in a dictionary, which is just an ordered list of words. Problems of this kind are called **searching problems**. We will discuss several algorithms for searching in this section. We will study the number of steps used by each of these algorithms in Section 3.3.

The general searching problem can be described as follows: Locate an element  $x$  in a list of distinct elements  $a_1, a_2, \dots, a_n$ , or determine that it is not in the list. The solution to this search problem is the location of the term in the list that equals  $x$  (that is,  $i$  is the solution if  $x = a_i$ ) and is 0 if  $x$  is not in the list.

**THE LINEAR SEARCH** The first algorithm that we will present is called the **linear search**, or **sequential search**, algorithm. The linear search algorithm begins by comparing  $x$  and  $a_1$ . When  $x = a_1$ , the solution is the location of  $a_1$ , namely, 1. When  $x \neq a_1$ , compare  $x$  with  $a_2$ . If  $x = a_2$ , the solution is the location of  $a_2$ , namely, 2. When  $x \neq a_2$ , compare  $x$  with  $a_3$ . Continue this process, comparing  $x$  successively with each term of the list until a match is found, where the solution is the location of that term, unless no match occurs. If the entire list has been searched without locating  $x$ , the solution is 0. The pseudocode for the linear search algorithm is displayed as Algorithm 2.



### ALGORITHM 2 The Linear Search Algorithm.

```

procedure linear search( $x$ : integer,  $a_1, a_2, \dots, a_n$ : distinct integers)
 $i := 1$ 
while ( $i \leq n$  and  $x \neq a_i$ )
     $i := i + 1$ 
if  $i \leq n$  then  $location := i$ 
else  $location := 0$ 
return  $location$  { $location$  is the subscript of the term that equals  $x$ , or is 0 if  $x$  is not found}

```

**THE BINARY SEARCH** We will now consider another searching algorithm. This algorithm can be used when the list has terms occurring in order of increasing size (for instance: if the terms are numbers, they are listed from smallest to largest; if they are words, they are listed in lexicographic, or alphabetic, order). This second searching algorithm is called the **binary search algorithm**. It proceeds by comparing the element to be located to the middle term of the list. The list is then split into two smaller sublists of the same size, or where one of these smaller lists has one fewer term than the other. The search continues by restricting the search to the appropriate sublist based on the comparison of the element to be located and the middle term. In Section 3.3, it will be shown that the binary search algorithm is much more efficient than the linear search algorithm. Example 3 demonstrates how a binary search works.



**EXAMPLE 3** To search for 19 in the list

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22,

first split this list, which has 16 terms, into two smaller lists with eight terms each, namely,


1 2 3 5 6 7 8 10      12 13 15 16 18 19 20 22.

Then, compare 19 and the largest term in the first list. Because  $10 < 19$ , the search for 19 can be restricted to the list containing the 9th through the 16th terms of the original list. Next, split this list, which has eight terms, into the two smaller lists of four terms each, namely,

12 13 15 16      18 19 20 22.

Because  $16 < 19$  (comparing 19 with the largest term of the first list) the search is restricted to the second of these lists, which contains the 13th through the 16th terms of the original list. The list 18 19 20 22 is split into two lists, namely,

18 19      20 22.

Because 19 is not greater than the largest term of the first of these two lists, which is also 19, the search is restricted to the first list: 18 19, which contains the 13th and 14th terms of the original list. Next, this list of two terms is split into two lists of one term each: 18 and 19. Because  $18 < 19$ , the search is restricted to the second list: the list containing the 14th term of the list, which is 19. Now that the search has been narrowed down to one term, a comparison is made, and 19 is located as the 14th term in the original list. 

We now specify the steps of the binary search algorithm. To search for the integer  $x$  in the list  $a_1, a_2, \dots, a_n$ , where  $a_1 < a_2 < \dots < a_n$ , begin by comparing  $x$  with the middle term  $a_m$  of the list, where  $m = \lfloor (n + 1)/2 \rfloor$ . (Recall that  $\lfloor x \rfloor$  is the greatest integer not exceeding  $x$ .) If  $x > a_m$ , the search for  $x$  is restricted to the second half of the list, which is  $a_{m+1}, a_{m+2}, \dots, a_n$ . If  $x$  is not greater than  $a_m$ , the search for  $x$  is restricted to the first half of the list, which is  $a_1, a_2, \dots, a_m$ .

The search has now been restricted to a list with no more than  $\lceil n/2 \rceil$  elements. (Recall that  $\lceil x \rceil$  is the smallest integer greater than or equal to  $x$ .) Using the same procedure, compare  $x$  to the middle term of the restricted list. Then restrict the search to the first or second half of the list. Repeat this process until a list with one term is obtained. Then determine whether this term is  $x$ . Pseudocode for the binary search algorithm is displayed as Algorithm 3.

#### ALGORITHM 3 The Binary Search Algorithm.

```
procedure binary search ( $x$ : integer,  $a_1, a_2, \dots, a_n$ : increasing integers)
 $i := 1$  { $i$  is left endpoint of search interval}
 $j := n$  { $j$  is right endpoint of search interval}
while  $i < j$ 
     $m := \lfloor (i + j)/2 \rfloor$ 
    if  $x > a_m$  then  $i := m + 1$ 
    else  $j := m$ 
if  $x = a_i$  then  $location := i$ 
else  $location := 0$ 
return  $location$  { $location$  is the subscript  $i$  of the term  $a_i$  equal to  $x$ , or 0 if  $x$  is not found}
```

Algorithm 3 proceeds by successively narrowing down the part of the sequence being searched. At any given stage only the terms from  $a_i$  to  $a_j$  are under consideration. In other words,  $i$  and  $j$  are the smallest and largest subscripts of the remaining terms, respectively. Algorithm 3 continues narrowing the part of the sequence being searched until only one term of the sequence remains. When this is done, a comparison is made to see whether this term equals  $x$ .

## Sorting



Ordering the elements of a list is a problem that occurs in many contexts. For example, to produce a telephone directory it is necessary to alphabetize the names of subscribers. Similarly, producing a directory of songs available for downloading requires that their titles be put in alphabetic order. Putting addresses in order in an e-mail mailing list can determine whether there are duplicated addresses. Creating a useful dictionary requires that words be put in alphabetical order. Similarly, generating a parts list requires that we order them according to increasing part number.

Suppose that we have a list of elements of a set. Furthermore, suppose that we have a way to order elements of the set. (The notion of ordering elements of sets will be discussed in detail in Section 9.6.) **Sorting** is putting these elements into a list in which the elements are in increasing order. For instance, sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, 5, 7, 9. Sorting the list  $d, h, c, a, f$  (using alphabetical order) produces the list  $a, c, d, f, h$ .

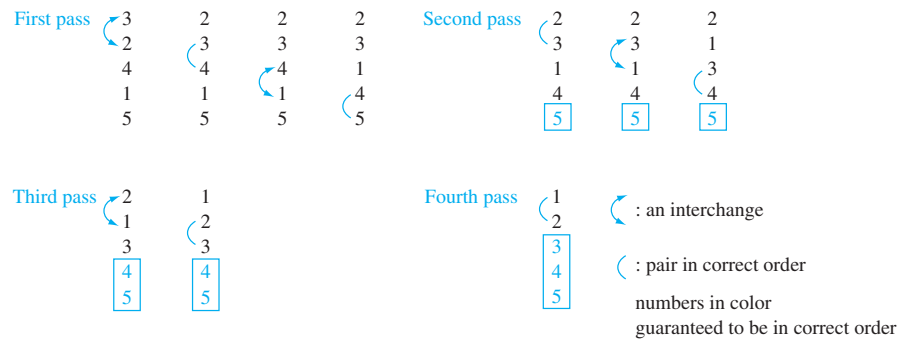
An amazingly large percentage of computing resources is devoted to sorting one thing or another. Hence, much effort has been devoted to the development of sorting algorithms. A surprisingly large number of sorting algorithms have been devised using distinct strategies, with new ones introduced regularly. In his fundamental work, *The Art of Computer Programming*, Donald Knuth devotes close to 400 pages to sorting, covering around 15 different sorting algorithms in depth! More than 100 sorting algorithms have been devised, and it is surprising how often new sorting algorithms are developed. Among the newest sorting algorithms that have caught on is the library sort, also known as the gapped insertion sort, invented as recently as 2006. There are many reasons why sorting algorithms interest computer scientists and mathematicians. Among these reasons are that some algorithms are easier to implement, some algorithms are more efficient (either in general, or when given input with certain characteristics, such as lists slightly out of order), some algorithms take advantage of particular computer architectures, and some algorithms are particularly clever. In this section we will introduce two sorting algorithms, the bubble sort and the insertion sort. Two other sorting algorithms, the selection sort and the binary insertion sort, are introduced in the exercises, and the shaker sort is introduced in the Supplementary Exercises. In Section 5.4 we will discuss the merge sort and introduce the quick sort in the exercises in that section; the tournament sort is introduced in the exercise set in Section 11.2. We cover sorting algorithms both because sorting is an important problem and because these algorithms can serve as examples for many important concepts.

Sorting is thought to hold the record as the problem solved by the most fundamentally different algorithms!



**THE BUBBLE SORT** The **bubble sort** is one of the simplest sorting algorithms, but not one of the most efficient. It puts a list into increasing order by successively comparing adjacent elements, interchanging them if they are in the wrong order. To carry out the bubble sort, we perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list, for a full pass. We iterate this procedure until the sort is complete. Pseudocode for the bubble sort is given as Algorithm 4. We can imagine the elements in the list placed in a column. In the bubble sort, the smaller elements “bubble” to the top as they are interchanged with larger elements. The larger elements “sink” to the bottom. This is illustrated in Example 4.

**EXAMPLE 4** Use the bubble sort to put 3, 2, 4, 1, 5 into increasing order.



**FIGURE 1** The Steps of a Bubble Sort.

**Solution:** The steps of this algorithm are illustrated in Figure 1. Begin by comparing the first two elements, 3 and 2. Because  $3 > 2$ , interchange 3 and 2, producing the list 2, 3, 4, 1, 5. Because  $3 < 4$ , continue by comparing 4 and 1. Because  $4 > 1$ , interchange 1 and 4, producing the list 2, 3, 1, 4, 5. Because  $4 < 5$ , the first pass is complete. The first pass guarantees that the largest element, 5, is in the correct position.

The second pass begins by comparing 2 and 3. Because these are in the correct order, 3 and 1 are compared. Because  $3 > 1$ , these numbers are interchanged, producing 2, 1, 3, 4, 5. Because  $3 < 4$ , these numbers are in the correct order. It is not necessary to do any more comparisons for this pass because 5 is already in the correct position. The second pass guarantees that the two largest elements, 4 and 5, are in their correct positions.

The third pass begins by comparing 2 and 1. These are interchanged because  $2 > 1$ , producing 1, 2, 3, 4, 5. Because  $2 < 3$ , these two elements are in the correct order. It is not necessary to do any more comparisons for this pass because 4 and 5 are already in the correct positions. The third pass guarantees that the three largest elements, 3, 4, and 5, are in their correct positions.

The fourth pass consists of one comparison, namely, the comparison of 1 and 2. Because  $1 < 2$ , these elements are in the correct order. This completes the bubble sort. ◀

#### ALGORITHM 4 The Bubble Sort.

```

procedure bubblesort( $a_1, \dots, a_n$  : real numbers with  $n \geq 2$ )
for  $i := 1$  to  $n - 1$ 
  for  $j := 1$  to  $n - i$ 
    if  $a_j > a_{j+1}$  then interchange  $a_j$  and  $a_{j+1}$ 
  { $a_1, \dots, a_n$  is in increasing order}
  
```



**THE INSERTION SORT** The **insertion sort** is a simple sorting algorithm, but it is usually not the most efficient. To sort a list with  $n$  elements, the insertion sort begins with the second element. The insertion sort compares this second element with the first element and inserts it before the first element if it does not exceed the first element and after the first element if it exceeds the first element. At this point, the first two elements are in the correct order. The third element is then compared with the first element, and if it is larger than the first element, it is compared with the second element; it is inserted into the correct position among the first three elements.

In general, in the  $j$ th step of the insertion sort, the  $j$ th element of the list is inserted into the correct position in the list of the previously sorted  $j - 1$  elements. To insert the  $j$ th element in the list, a linear search technique is used (see Exercise 43); the  $j$ th element is successively compared with the already sorted  $j - 1$  elements at the start of the list until the first element that



is not less than this element is found or until it has been compared with all  $j - 1$  elements; the  $j$ th element is inserted in the correct position so that the first  $j$  elements are sorted. The algorithm continues until the last element is placed in the correct position relative to the already sorted list of the first  $n - 1$  elements. The insertion sort is described in pseudocode in Algorithm 5.

**EXAMPLE 5** Use the insertion sort to put the elements of the list 3, 2, 4, 1, 5 in increasing order.

*Solution:* The insertion sort first compares 2 and 3. Because  $3 > 2$ , it places 2 in the first position, producing the list 2, 3, 4, 1, 5 (the sorted part of the list is shown in color). At this point, 2 and 3 are in the correct order. Next, it inserts the third element, 4, into the already sorted part of the list by making the comparisons  $4 > 2$  and  $4 > 3$ . Because  $4 > 3$ , 4 remains in the third position. At this point, the list is 2, 3, 4, 1, 5 and we know that the ordering of the first three elements is correct. Next, we find the correct place for the fourth element, 1, among the already sorted elements, 2, 3, 4. Because  $1 < 2$ , we obtain the list 1, 2, 3, 4, 5. Finally, we insert 5 into the correct position by successively comparing it to 1, 2, 3, and 4. Because  $5 > 4$ , it stays at the end of the list, producing the correct order for the entire list. ◀

#### ALGORITHM 5 The Insertion Sort.

```

procedure insertion sort( $a_1, a_2, \dots, a_n$ : real numbers with  $n \geq 2$ )
for  $j := 2$  to  $n$ 
     $i := 1$ 
    while  $a_j > a_i$ 
         $i := i + 1$ 
     $m := a_j$ 
    for  $k := 0$  to  $j - i - 1$ 
         $a_{j-k} := a_{j-k-1}$ 
     $a_i := m$ 
    { $a_1, \dots, a_n$  is in increasing order}

```

## Greedy Algorithms

Many algorithms we will study in this book are designed to solve **optimization problems**. The goal of such problems is to find a solution to the given problem that either minimizes or maximizes the value of some parameter. Optimization problems studied later in this text include finding a route between two cities with smallest total mileage, determining a way to encode messages using the fewest bits possible, and finding a set of fiber links between network nodes using the least amount of fiber.

Surprisingly, one of the simplest approaches often leads to a solution of an optimization problem. This approach selects the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution. Algorithms that make what seems to be the “best” choice at each step are called **greedy algorithms**. Once we know that a greedy algorithm finds a feasible solution, we need to determine whether it has found an optimal solution. (Note that we call the algorithm “greedy” whether or not it finds an optimal solution.) To do this, we either prove that the solution is optimal or we show that there is a counterexample where the algorithm yields a nonoptimal solution. To make these concepts more concrete, we will consider an algorithm that makes change using coins.

“Greedy is good ... Greedy is right, greed works. Greedy clarifies ...” – spoken by the character Gordon Gecko in the film *Wall Street*.



You have to prove that a greedy algorithm always finds an optimal solution.



**EXAMPLE 6** Consider the problem of making  $n$  cents change with quarters, dimes, nickels, and pennies, and using the least total number of coins. We can devise a greedy algorithm for making change for  $n$  cents by making a locally optimal choice at each step; that is, at each step we choose the coin of the largest denomination possible to add to the pile of change without exceeding  $n$  cents. For example, to make change for 67 cents, we first select a quarter (leaving 42 cents). We next select a second quarter (leaving 17 cents), followed by a dime (leaving 7 cents), followed by a nickel (leaving 2 cents), followed by a penny (leaving 1 cent), followed by a penny. ◀



We display a greedy change-making algorithm for  $n$  cents, using any set of denominations of coins, as Algorithm 6.

**ALGORITHM 6 Greedy Change-Making Algorithm.**

```

procedure change( $c_1, c_2, \dots, c_r$ : values of denominations of coins, where
     $c_1 > c_2 > \dots > c_r$ ;  $n$ : a positive integer)
  for  $i := 1$  to  $r$ 
     $d_i := 0$  { $d_i$  counts the coins of denomination  $c_i$  used}
    while  $n \geq c_i$ 
       $d_i := d_i + 1$  {add a coin of denomination  $c_i$ }
       $n := n - c_i$ 
  { $d_i$  is the number of coins of denomination  $c_i$  in the change for  $i = 1, 2, \dots, r$ }

```

We have described a greedy algorithm for making change using any finite set of coins with denominations  $c_1, c_2, \dots, c_r$ . In the particular case where the four denominations are quarters, dimes, nickels, and pennies, we have  $c_1 = 25$ ,  $c_2 = 10$ ,  $c_3 = 5$ , and  $c_4 = 1$ . For this case, we will show that this algorithm leads to an optimal solution in the sense that it uses the fewest coins possible. Before we embark on our proof, we show that there are sets of coins for which the greedy algorithm (Algorithm 6) does not necessarily produce change using the fewest coins possible. For example, if we have only quarters, dimes, and pennies (and no nickels) to use, the greedy algorithm would make change for 30 cents using six coins—a quarter and five pennies—whereas we could have used three coins, namely, three dimes.

**LEMMA 1** If  $n$  is a positive integer, then  $n$  cents in change using quarters, dimes, nickels, and pennies using the fewest coins possible has at most two dimes, at most one nickel, at most four pennies, and cannot have two dimes and a nickel. The amount of change in dimes, nickels, and pennies cannot exceed 24 cents.

**Proof:** We use a proof by contradiction. We will show that if we had more than the specified number of coins of each type, we could replace them using fewer coins that have the same value. We note that if we had three dimes we could replace them with a quarter and a nickel, if we had two nickels we could replace them with a dime, if we had five pennies we could replace them with a nickel, and if we had two dimes and a nickel we could replace them with a quarter. Because we can have at most two dimes, one nickel, and four pennies, but we cannot have two dimes and a nickel, it follows that 24 cents is the most money we can have in dimes, nickels, and pennies when we make change using the fewest number of coins for  $n$  cents. ◀

**THEOREM 1** The greedy algorithm (Algorithm 6) produces change using the fewest coins possible.

**Proof:** We will use a proof by contradiction. Suppose that there is a positive integer  $n$  such that there is a way to make change for  $n$  cents using quarters, dimes, nickels, and pennies that uses fewer coins than the greedy algorithm finds. We first note that  $q'$ , the number of quarters used in this optimal way to make change for  $n$  cents, must be the same as  $q$ , the number of quarters used by the greedy algorithm. To show this, first note that the greedy algorithm uses the most quarters possible, so  $q' \leq q$ . However, it is also the case that  $q'$  cannot be less than  $q$ . If it were, we would need to make up at least 25 cents from dimes, nickels, and pennies in this optimal way to make change. But this is impossible by Lemma 1.

Because there must be the same number of quarters in the two ways to make change, the value of the dimes, nickels, and pennies in these two ways must be the same, and these coins are worth no more than 24 cents. There must be the same number of dimes, because the greedy algorithm used the most dimes possible and by Lemma 1, when change is made using the fewest coins possible, at most one nickel and at most four pennies are used, so that the most dimes possible are also used in the optimal way to make change. Similarly, we have the same number of nickels and, finally, the same number of pennies. ◀

A greedy algorithm makes the best choice at each step according to a specified criterion. The next example shows that it can be difficult to determine which of many possible criteria to choose.

**EXAMPLE 7** Suppose we have a group of proposed talks with preset start and end times. Devise a greedy algorithm to schedule as many of these talks as possible in a lecture hall, under the assumptions that once a talk starts, it continues until it ends, no two talks can proceed at the same time, and a talk can begin at the same time another one ends. Assume that talk  $j$  begins at time  $s_j$  (where  $s$  stands for *start*) and ends at time  $e_j$  (where  $e$  stands for *end*).

**Solution:** To use a greedy algorithm to schedule the most talks, that is, an optimal schedule, we need to decide how to choose which talk to add at each step. There are many criteria we could use to select a talk at each step, where we chose from the talks that do not overlap talks already selected. For example, we could add talks in order of earliest start time, we could add talks in order of shortest time, we could add talks in order of earliest finish time, or we could use some other criterion.

We now consider these possible criteria. Suppose we add the talk that starts earliest among the talks compatible with those already selected. We can construct a counterexample to see that the resulting algorithm does not always produce an optimal schedule. For instance, suppose that we have three talks: Talk 1 starts at 8 A.M. and ends at 12 noon, Talk 2 starts at 9 A.M. and ends at 10 A.M., and Talk 3 starts at 11 A.M. and ends at 12 noon. We first select the Talk 1 because it starts earliest. But once we have selected Talk 1 we cannot select either Talk 2 or Talk 3 because both overlap Talk 1. Hence, this greedy algorithm selects only one talk. This is not optimal because we could schedule Talk 2 and Talk 3, which do not overlap.

Now suppose we add the talk that is shortest among the talks that do not overlap any of those already selected. Again we can construct a counterexample to show that this greedy algorithm does not always produce an optimal schedule. So, suppose that we have three talks: Talk 1 starts at 8 A.M. and ends at 9:15 A.M., Talk 2 starts at 9 A.M. and ends at 10 A.M., and Talk 3 starts at 9:45 A.M. and ends at 11 A.M. We select Talk 2 because it is shortest, requiring one hour. Once we select Talk 2, we cannot select either Talk 1 or Talk 3 because neither is compatible with Talk 2. Hence, this greedy algorithm selects only one talk. However, it is possible to select two talks, Talk 1 and Talk 3, which are compatible.

However, it can be shown that we schedule the most talks possible if in each step we select the talk with the earliest ending time among the talks compatible with those already selected. We will prove this in Chapter 5 using the method of mathematical induction. The first step we will make is to sort the talks according to increasing finish time. After this sorting, we relabel the talks so that  $e_1 \leq e_2 \leq \dots \leq e_n$ . The resulting greedy algorithm is given as Algorithm 7. ◀

**ALGORITHM 7 Greedy Algorithm for Scheduling Talks.**

```

procedure schedule( $s_1 \leq s_2 \leq \dots \leq s_n$ : start times of talks,
 $e_1 \leq e_2 \leq \dots \leq e_n$ : ending times of talks)
  sort talks by finish time and reorder so that  $e_1 \leq e_2 \leq \dots \leq e_n$ 
   $S := \emptyset$ 
  for  $j := 1$  to  $n$ 
    if talk  $j$  is compatible with  $S$  then
       $S := S \cup \{\text{talk } j\}$ 
  return  $S$  { $S$  is the set of talks scheduled}

```

## The Halting Problem



We will now describe a proof of one of the most famous theorems in computer science. We will show that there is a problem that cannot be solved using any procedure. That is, we will show there are unsolvable problems. The problem we will study is the **halting problem**. It asks whether there is a procedure that does this: It takes as input a computer program and input to the program and determines whether the program will eventually stop when run with this input. It would be convenient to have such a procedure, if it existed. Certainly being able to test whether a program entered into an infinite loop would be helpful when writing and debugging programs. However, in 1936 Alan Turing showed that no such procedure exists (see his biography in Section 13.4).

Before we present a proof that the halting problem is unsolvable, first note that we cannot simply run a program and observe what it does to determine whether it terminates when run with the given input. If the program halts, we have our answer, but if it is still running after any fixed length of time has elapsed, we do not know whether it will never halt or we just did not wait long enough for it to terminate. After all, it is not hard to design a program that will stop only after more than a billion years has elapsed.

We will describe Turing's proof that the halting problem is unsolvable; it is a proof by contradiction. (The reader should note that our proof is not completely rigorous, because we have not explicitly defined what a procedure is. To remedy this, the concept of a Turing machine is needed. This concept is introduced in Section 13.5.)

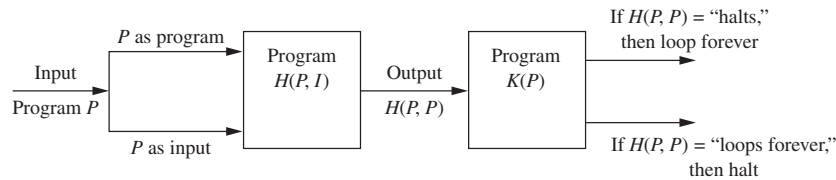


**Proof:** Assume there is a solution to the halting problem, a procedure called  $H(P, I)$ . The procedure  $H(P, I)$  takes two inputs, one a program  $P$  and the other  $I$ , an input to the program  $P$ .  $H(P, I)$  generates the string “halt” as output if  $H$  determines that  $P$  stops when given  $I$  as input. Otherwise,  $H(P, I)$  generates the string “loops forever” as output. We will now derive a contradiction.

When a procedure is coded, it is expressed as a string of characters; this string can be interpreted as a sequence of bits. This means that a program itself can be used as data. Therefore a program can be thought of as input to another program, or even itself. Hence,  $H$  can take a program  $P$  as both of its inputs, which are a program and input to this program.  $H$  should be able to determine whether  $P$  will halt when it is given a copy of itself as input.

To show that no procedure  $H$  exists that solves the halting problem, we construct a simple procedure  $K(P)$ , which works as follows, making use of the output  $H(P, P)$ . If the output of  $H(P, P)$  is “loops forever,” which means that  $P$  loops forever when given a copy of itself as input, then  $K(P)$  halts. If the output of  $H(P, P)$  is “halt,” which means that  $P$  halts when given a copy of itself as input, then  $K(P)$  loops forever. That is,  $K(P)$  does the opposite of what the output of  $H(P, P)$  specifies. (See Figure 2.)

Now suppose we provide  $K$  as input to  $K$ . We note that if the output of  $H(K, K)$  is “loops forever,” then by the definition of  $K$  we see that  $K(K)$  halts. Otherwise, if the output of  $H(K, K)$



**FIGURE 2** Showing that the Halting Problem is Unsolvable.

is “halt,” then by the definition of  $K$  we see that  $K(K)$  loops forever, in violation of what  $H$  tells us. In both cases, we have a contradiction.

Thus,  $H$  cannot always give the correct answers. Consequently, there is no procedure that solves the halting problem. ◀

## Exercises

- List all the steps used by Algorithm 1 to find the maximum of the list 1, 8, 12, 9, 11, 2, 14, 5, 10, 4.
- Determine which characteristics of an algorithm described in the text (after Algorithm 1) the following procedures have and which they lack.
  - procedure** *double*( $n$ : positive integer)  
   **while**  $n > 0$   
      $n := 2n$
  - procedure** *divide*( $n$ : positive integer)  
   **while**  $n \geq 0$   
      $m := 1/n$   
      $n := n - 1$
  - procedure** *sum*( $n$ : positive integer)  
    $sum := 0$   
   **while**  $i < 10$   
      $sum := sum + i$
  - procedure** *choose*( $a, b$ : integers)  
    $x := \text{either } a \text{ or } b$
- Devise an algorithm that finds the sum of all the integers in a list.
- Describe an algorithm that takes as input a list of  $n$  integers and produces as output the largest difference obtained by subtracting an integer in the list from the one following it.
- Describe an algorithm that takes as input a list of  $n$  integers in nondecreasing order and produces the list of all values that occur more than once. (Recall that a list of integers is **nondecreasing** if each integer in the list is at least as large as the previous integer in the list.)
- Describe an algorithm that takes as input a list of  $n$  integers and finds the number of negative integers in the list.
- Describe an algorithm that takes as input a list of  $n$  integers and finds the location of the last even integer in the list or returns 0 if there are no even integers in the list.
- Describe an algorithm that takes as input a list of  $n$  distinct integers and finds the location of the largest even integer in the list or returns 0 if there are no even integers in the list.
- A **palindrome** is a string that reads the same forward and backward. Describe an algorithm for determining whether a string of  $n$  characters is a palindrome.
- Devise an algorithm to compute  $x^n$ , where  $x$  is a real number and  $n$  is an integer. [Hint: First give a procedure for computing  $x^n$  when  $n$  is nonnegative by successive multiplication by  $x$ , starting with 1. Then extend this procedure, and use the fact that  $x^{-n} = 1/x^n$  to compute  $x^n$  when  $n$  is negative.]
- Describe an algorithm that interchanges the values of the variables  $x$  and  $y$ , using only assignments. What is the minimum number of assignment statements needed to do this?
- Describe an algorithm that uses only assignment statements that replaces the triple  $(x, y, z)$  with  $(y, z, x)$ . What is the minimum number of assignment statements needed?
- List all the steps used to search for 9 in the sequence 1, 3, 4, 5, 6, 8, 9, 11 using
  - a linear search.
  - a binary search.
- List all the steps used to search for 7 in the sequence given in Exercise 13 for both a linear search and a binary search.
- Describe an algorithm that inserts an integer  $x$  in the appropriate position into the list  $a_1, a_2, \dots, a_n$  of integers that are in increasing order.
- Describe an algorithm for finding the smallest integer in a finite sequence of natural numbers.
- Describe an algorithm that locates the first occurrence of the largest element in a finite list of integers, where the integers in the list are not necessarily distinct.
- Describe an algorithm that locates the last occurrence of the smallest element in a finite list of integers, where the integers in the list are not necessarily distinct.