
SEARCHING AND SORTING

6:1 INTRODUCTION RECORD KEEPING

A college's financial aid office has just created the job of Director of Student Employment. The responsibilities of this position include the organization of student employment information. Until now this information has been kept in the following fashion. Each student employee has been assigned a record card on which is written payroll information, including the student's social security number. These record cards are organized in a file drawer arranged in alphabetical order of the students' last names. Each time the treasurer's office issues a payroll check the director receives a memo containing the payee's social security number, the total amount of the check, and the amount withheld for various taxes. Of course, she wishes that these memos also contained the payee's name; however, the particular computer program that the treasurer's office uses to cut checks doesn't have that capability. When a payroll memo arrives, the director examines each record card in turn to determine if the social security number on the card is identical to the number on the memo.

Question 1.1. Suppose that there were 20 cards in the director's file drawer. (a) When a payroll memo arrives, what is the minimum number of cards that the director might have to check? (b) What is the maximum number of cards that the director might have to check? (c) About how many cards (on average) would you expect the director to have to check?

Question 1.2. Suppose that each of the 20 students whose cards are in the file drawer receives exactly one payroll check each week. (a) What is the total number of social security number comparisons that the director will have to make to

6 SEARCHING AND SORTING

record all of the payroll transactions? (b) If it takes 2 seconds to make a comparison and 1 minute to record all the information on a file card, will the director spend more time making comparisons or recording information?

Now let's answer the previous questions if there are n record cards in the director's drawer. The minimum possible number of comparisons occurs when the payee happens to be the individual whose card is first in the file, the one whose name is alphabetical] y first. In this instance there is just one comparison to make. The largest number of comparisons occurs when the payee is the individual whose card is last in the file. In this case there would be n comparisons to make. It is plausible to think that the average number of comparisons should be the average of the smallest number and the largest number. Here that number would be $(n + 1)/2$. In fact, this is correct as we see by the following explicit computation.

If every individual in the file is paid exactly once, we can count the total number of comparisons in the following manner. First, note that the payee who is listed first alphabetically will require just one comparison to locate. We don't know which memo corresponds to this first payee, but whichever one it is, it will still take just one comparison. Similarly, the payee who is listed second alphabetically will take exactly two comparisons to locate. In general, the payee who is listed k th alphabetically will take exactly k comparisons to locate (regardless of when this memo is processed). Thus the total number of comparisons will be

$$1 + 2 + 3 + \cdots + k + \cdots + n = \frac{n(n + 1)}{2}.$$

Since the total number of comparisons needed is $n(n + 1)/2$, the average number of comparisons needed will be this total divided by the number of payees. This yields $(n + 1)/2$ comparisons on average. The total time needed for comparisons will be $n(n + 1)$ seconds while the time required to record the payroll information will be $60n$ seconds. Thus if $n \geq 60$, more time will be spent finding the correct file than writing information to it.

Let's formalize the director's task.

Problem. Given an array $A = (a_1, a_2, \dots, a_n)$ and an object S , determine S 's position in A , that is, find an index i such that $a_i = S$ (if such an i exists).

Algorithm SEQSEARCH

- STEP 1. Input A and S .
- STEP 2. For $i = 1$ to n do
 - STEP 3. If $a_i = S$, then output i and stop.
- STEP 4. Output "S not in A " and stop.

If we count the comparisons in step 3, then the worst case will occur either if S is not in A or if $S = a_n$. In this instance SEQSEARCH requires n comparisons. Thus the complexity of this algorithm is $O(n)$. Note that in our particular example with social security numbers, S and the elements in the array A are numbers; however, all that is required for this algorithm to work is that we can determine whether $a_i = S$. Thus SEQSEARCH would work equally well when the entries of A are words.

The director decides that record keeping would be more efficient if the record cards were kept in order of their social security numbers. The director begins the sorting process by finding the card with the smallest social security number. She does this by comparing the number on the first card with the number on the second. She keeps the smaller of the two and then compares it with the number on the third card. She picks the smaller and now has the smallest number from the first three cards.

Question 1.3. In a drawer of 20 record cards, how many comparisons would be required to be certain of finding the card with the smallest social security number?

We formalize the problem and the response.

Problem. Given an array of numbers $A = (a_1, a_2, \dots, a_n)$, sort these numbers into increasing order, that is, arrange the numbers within the array so that $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$.

Algorithm SELECTSORT

```

STEP 1. Input  $A$ , an array of  $n$  numbers
STEP 2. For  $i = 1$  to  $n - 1$  do {Find the correct  $i$ th number.}
    Begin
    STEP 3. Set  $TN := a_i$  { $TN$  = temporary number}
    STEP 4. For  $j = i + 1$  to  $n$  do
        STEP 5. If  $a_j < TN$ , switch  $a_j$  and  $TN$ 
    STEP 6. Set  $a_i := TN$ 
    End {Step 2}
STEP 7. Output  $A$  and stop.

```

Example 1.1. Table 6.1 gives a trace of SELECTSORT applied to the array $A = (4, 7, 3)$. Notice that all the action occurs at step 5.

Since the smallest remaining element is repeatedly selected, this method is called Selection sort. See Exercises 11 to 13 for a comparison with the sorting algorithm known as **Bubblesort**.

Table 6.1

Step No.	i	j	a_1	a_2	a_3	TN
3	1	?	4	7	3	4
5	1	2	4	7	3	4
5	1	3	4	7	4	3
6	1	3	3	7	4	3
3	2	3	3	7	4	7
5	2	3	3	7	7	4
6	2	3	3	4	7	4

Question 1.4. Apply SELECTSORT to the array $A = (6,4,2, 3)$. Exhibit the values assigned to i , j , TN, and each location in A after every execution of step 5.

Theorem 1.1. SELECTSORT is a $O(n^2)$ algorithm.

Proof. We count the comparisons, which only occur in step 5. When i is assigned the value 1, j varies from 2 to n . Thus there are $n - 1$ different values assigned to j and $n - 1$ comparisons when $i = 1$. When i is assigned the value 2, j varies from 3 to n . Thus there are $n - 2$ comparisons. For general i , j varies from $i + 1$ to n . In this case there are $(n - (i + 1) + 1) = n - i$ comparisons. Hence the total number of comparisons equals

$$(n - 1) + (n - 2) + \cdots + (n - i) + \cdots + 1 = \frac{n(n - 1)}{2} = O(n^2). \quad \bullet$$

A bit analysis of SELECTSORT would begin by noting that each of the n numbers in the input array could be represented by M bits. Thus the total input size would be nM . Every comparison of two M bit numbers would require, in the worst case, M bit comparisons. Thus the total number of bit comparisons would be $Mn(n - 1)/2$. If M is constant, then SELECTSORT is quadratic in the bit analysis also.

Notice that SELECTSORT could work equally well on arrays of words using alphabetical ordering. In a subsequent section we shall see that SELECTSORT can operate on sets with more general orderings. We'll also find that there are more efficient algorithms to perform sorting as well as searching however, for small arrays SEQSEARCH and SELECTSORT are worth using, in part because they are so simple.

Here is the terminology we shall use throughout the rest of the chapter. Each unit of information to be sorted is called a **record**. The set of records is called a file. The element in the record with which the sorting is done is called the **key**.

Thus in the employment director's office, her drawer contains the file. Each card in the file is a record and the key is the social security number on the card. To keep numerical examples simple, we shall often consider a record that consists only of the key, but in applications the record will contain more information. Consequently, interchanging two records in a file will be a more time-consuming process than that of switching two numbers. If the records are stored in computer memory and accessed by a language that admits the use of pointers, then the pointers will be changed rather than the records.

EXERCISES FOR SECTION 1

1. Apply SEQSEARCH to the following arrays and objects S ; record the output of the algorithm.
 - (a) $A = \langle 1, 2, 3, \dots, 17 \rangle$, $S = 15$.
 - (b) $A = (1, 2, 3, \dots, 17)$, $S = 12.5$.
 - (c) $A = \langle \text{apple, banana, cantaloupe, kiwi, mango, papaya} \rangle$, $S = \text{strawberry}$.
 - (d) $A = \langle a, b, c, \dots, z \rangle$, $S = h$.
 - (e) $A = \langle a, b, c, \dots, z \rangle$, $S = \&$.
 - (f) $A = \langle 1, 2, 3, a, b, c, \#, \$, \%, \hat{\ } \rangle$, $S = \$$.

(Note: In Exercises 2 to 7 we assume that, as in Questions 1.1 to 1.3, the record cards are listed alphabetically and the payroll memos come identified by social security number.)

2. Suppose that there are 40 student employees who each receive 2 checks per month. How many comparisons does the director make in a month using SEQSEARCH? If it takes 2 seconds to make a comparison and 1 minute to record the payroll information, which requires more time, comparing or recording information?
3. Suppose that there are 20 student employees and exactly 10 receive a check in any given week. What is the minimum and maximum number of comparisons that the director might make in a week?
4. Suppose that there are n student employees who each receive k checks per month. How many comparisons will the director make in one month?
5. Suppose that there are $2n$ student employees and that exactly n of these students receive a check in a given week. What is the minimum and maximum number of comparisons that might be performed? What can be said about the average number of comparisons that will be made?
6. Suppose that there are n student employees who each receive one check per week. If it takes 3 seconds to make a comparison and 30 seconds to record the

salary information, for what values of n is more time spent on comparisons than on recording?

7. Suppose that there are n student employees who each receive one check per week. If it takes x seconds to make a comparison and y seconds to record information, then for what values of x and y do comparing and recording take the same amount of time? For what values of x and y does comparing take more time than recording?
8. Apply SELECTSORT to the arrays $(1, 2, 3)$, $\langle 3, 2, 1 \rangle$, and $(3, 1, 2, 1)$. Trace out the values assigned to i, j , TN, and every location in A after each execution of step 5.
9. Write an algorithm that, given an array of numbers, (a) selects the largest number and places it in the last position, (b) selects the next largest number and places it in the next to last position, and (c), in general, finds the largest remaining number and places it in the last unfilled position. Analyze the complexity of your algorithm.
10. Write an algorithm that finds the largest and the smallest entry in $A = \langle a_1, a_2, \dots, a_n \rangle$, an array of real numbers. Count the number of comparisons made in the worst case.
11. Look back at the algorithm BUBBLES, Exercise 2.4.13. Recall that this algorithm found the largest entry in an array of n elements and placed it in the last location. BUBBLES can be readily transformed into a procedure that can be repeatedly called to sort the entire array. Here is an algorithm that does just this.

Algorithm BUBBLESORT

STEP 1. Input m , a positive integer, and the array $X = \langle x_1, \dots, x_m \rangle$

STEP 2. For $n = m$ down to 2 do

STEP 3. Call BUBBLES (n, x_1, \dots, x_n)

STEP 4. Output $\langle x_1, x_2, \dots, x_m \rangle$ and stop.

Apply BUBBLESORT to the following arrays, exhibiting the values of the array, n and j (the index in BUBBLES) throughout.

(a) $A = \langle 4, 7, 3 \rangle$.

(b) $B = \langle 2, 1, 4, 3, 6, 5 \rangle$.

(c) $C = \langle 4, 3, 2, 1 \rangle$.

12. Count the number of comparisons made by BUBBLESORT. Compare the number of comparisons made in BUBBLESORT and SELECTSORT. Is one algorithm more efficient than the other?
13. How might you modify BUBBLESORT to recognize when the array X was already in order?

14. One way a record can contain more than the key, is using a 2-dimensional array $A = \langle a_{i,j}; i = 1, \dots, m, j = 1, \dots, n \rangle$. This can be pictured as a matrix with m rows and n columns:

$$\begin{array}{ccccccc}
 a_{1,1} & a_{1,2} & \cdots & a_{1,j} & \cdots & a_{1,n} & \\
 a_{2,1} & a_{2,2} & \cdots & a_{2,j} & \cdots & a_{2,n} & \\
 \cdots & & & & & & \\
 a_{i,1} & a_{i,2} & \cdots & a_{i,j} & \cdots & a_{i,n} & \\
 \cdots & & & & & & \\
 a_{m,1} & a_{m,2} & \cdots & a_{m,j} & \cdots & a_{m,n} &
 \end{array}$$

Each row might represent the record of one student, and different columns contain different types of information. Suppose that the key for each record is stored in the first column so that the key for i th record is the entry $a_{i,1}$ for $i = 1, 2, \dots, m$. Use the idea of SELECTSORT to design an algorithm to sort the array A so that the rows of A are rearranged to have their first entries (the keys) listed in increasing order $a_{1,1} \leq a_{2,1} \leq \dots \leq a_{m,1}$. How many comparisons does the algorithm use? In the worst case how many assignment statements are there? Your answers will depend on n and m .

15. Write an algorithm that finds the second smallest entry in an array $A = \langle a_1, a_2, \dots, a_n \rangle$ of real numbers. Count the number of comparisons made.
16. Here is the idea for an algorithm to find the k th smallest entry in an array A of n numbers: Find the smallest entry of A , then find the second smallest entry, and so on, until the k th smallest entry is found. Write an algorithm that implements this idea and count the number of comparisons; your answer will be in terms of n and k .
17. Here is another algorithmic solution to the problem of finding the k th smallest entry in an array A (see Exercise 16) First order the array using SELECTSORT and then find the k th entry of the sorted array. Compare the number of comparisons made by this algorithm with that of Exercise 16, which is more efficient?
18. Suppose that you have a balance scale with which you can determine which (if either) of a pair of given coins is lighter in weight. Given n supposedly identical coins, but such that one weighs less than the others, give a technique suggested by SELECTSORT to find the light coin. How many comparisons will your technique require in the worst case?
19. Suppose that you have 16 supposedly identical coins, exactly one of which weighs less. Using a balance scale, each pan of which can hold as many coins as you like, how can you find the light coin with only 4 weighings?

6:2 SEARCHING A SORTED FILE

We return to the employment director's problem of transcribing payroll information. We assume that there are n employees whose record cards are filed now in the order of increasing social security number. When a memo arrives from the payroll office, the director searches for the record whose social security number is the same as the one on the memo. Suppose she selects the m th record from the file, or drawer, and compares the two social security numbers. If the two numbers are equal, then the director writes the information on the selected record. If the number on the memo is less than the number on the m th record, then the correct record must be located in the front portion of the file. Otherwise, the correct record must be located behind the m th record.

Of course, the director hopes to pick the correct record on the first try. However, she does not believe in her own good luck. Furthermore (with a touch of pessimism), the director believes that when she picks a record to compare with the payroll memo, the record she really wants will be in the larger part of the remaining records. Thus the director wants to choose a record in the m th position so that there are about as many records in front of the m th record as there are behind the m th record. If the drawer has n records, the director picks the record roughly in the middle, the record in the m th position, where $m = \lfloor (n + 1)/2 \rfloor$. The director has, of course, assumed a worst-case scenario.

Question 2.1. Find $m = \lfloor (n + 1)/2 \rfloor$ if $n = 136, 68, 34, 17, 9, 5,$ and 3 .

Question 2.2. If the drawer contains n records and the m th record is selected, where $m = \lfloor (n + 1)/2 \rfloor$, when is it the case that there are exactly the same number of records before and after the m th record? When these two numbers differ, by how much do they differ? After examining this m th record, what is the largest number of records that still must be searched?

Let's assume for the moment that the director has selected m and the number on the memo is less than the number on the m th record. Then she begins the search all over again, confining her attention to that portion of the file that is in front of the m th record. In pseudocode she sets $n := m - 1$ and then chooses m to be (as before) $\lfloor (n + 1)/2 \rfloor$. On the other hand, if the number on the memo is greater than the number on the m th record, then the correct record must be in position j , where $(m + 1) \leq j \leq n$. As above she begins the search all over again, concentrating on the records in positions $m + 1, \dots, n$. The next record to select is the one that, as nearly as possible, divides the remaining records into equal piles.

Question 2.3. For the following pairs (i, j) find the number that will be the index of the entry that, as nearly as possible, divides $\langle a_i, \dots, a_j \rangle$ into two equal pieces: $(6, 8), (10, 17), (18, 33), (35, 67),$ and $(69, 136)$.

In general, as the director progresses, she narrows down the possible records that might correspond with the memo to a subarray $\langle a_i, a_{i+1}, \dots, a_j \rangle$ of the original array A . She wants to select the “middle” record of this subarray. The index of the “middle” record is essentially the average of the indices of the end records. We say essentially because the average might not be an integer. However, a record that, as nearly as possible, divides the subarray into two equal pieces has index $m = \lfloor (i + j)/2 \rfloor$. With this insight we can now formulate the director’s algorithm.

Problem. Given an array $A = \langle a_1, a_2, \dots, a_n \rangle$ whose elements are numbers listed in increasing order and a number S , determine S ’s position in A , that is, find an index i (if it exists) such that $a_i = S$.

Algorithm **BINAR YSEARCH**

- STEP 1. Input A , an array of n numbers in increasing order, and a number S
 STEP 2. Set $first := 1$, $last := n$
 STEP 3. While $first \leq last$ do
 Begin
 STEP 4. Set $mid := \lfloor (first + last)/2 \rfloor$
 STEP 5. If $S = a_{mid}$, then output “found S at location mid and stop.
 STEP 6. If $S < a_{mid}$, then set $last := mid - 1$,
 Else set $first := mid + 1$
 End {Step 3}
 STEP 7. Output “ S is not in A ” and stop.

Note that in step 6 exactly one of two assignment statements is executed, depending on the result of the comparison in that step.

Example 2.1. Table 6.2 is a trace of **BINARYSEARCH**, where $A = (3, 4, 6, 7, 9, 11)$ and $S = 9$. We begin after the first encounter with step 4.

Table 6.2

<i>Step No.</i>	<i>first</i>	<i>last</i>	<i>mid</i>	a_{mid}
4	1	6	3	6
5	1	6	3	6
6	4	6	3	6
4	4	6	5	9
5	4	6	5	9

Question 2.4. Trace **BINARYSEARCH** if A consists of the first eight primes in increasing order and (a) $S = 5$, (b) $S = 10$, and (c) $S = 17$. In each case how many elements in the array do you examine?

BINARYSEARCH can find S without examining all the entries in A because the elements of A are numbers listed in increasing order. Actually, this algorithm will work on any set that is totally ordered. See Exercises 4.6.12 to 4.6.14. Since A is totally ordered, either $S < a_{\text{mid}}$ or $a_{\text{mid}} \leq S$. Consequently, the value of (last-first) decreases with each loop and so BINARYSEARCH must terminate. Furthermore, the transitive property allows the algorithm to check S against a_{mid} and discard about half of the ordered list at each pass through the loop. Exercise 6 asks you to modify BINARYSEARCH so that it works on the set of all English words in alphabetical order.

Theorem 2.1. BINARYSEARCH requires at most $3 \lfloor \log(n) \rfloor + 4$ comparisons to search an ordered array of n numbers.

Proof. First note that each of steps 3, 5, and 6 requires exactly one comparison. Thus each time we execute the loop beginning at step 3, we use no more than three comparisons. The proof will be by induction on the number of elements in the array. We begin with the base case $n = 1$. Given the array $A = \langle a \rangle$, the algorithm uses two comparisons if $S = a$. If $S \neq a$, then the algorithm cycles through the loop once and executes step 3 one additional time. Thus a total of four comparisons is needed in this case.

The inductive hypothesis will be that BINARYSEARCH can search any ordered array of t elements with at most $3 \lfloor \log(t) \rfloor + 4$ comparisons for any $t < n$. We suppose that A is an ordered array with n elements. If we find equality the first time at step 5, we are done, using 2 comparisons. Otherwise, we return to step 3 with a smaller array, having performed three comparisons. The new array contains no more than half of the elements of the original array. (See Question 2.2.) By the inductive hypothesis it takes at most $3 \lfloor \log(n/2) \rfloor + 4$ comparisons to search the new array. Thus the total number of comparisons needed to search the original array is at most

$$\begin{aligned} 3 + (3 \lfloor \log(n/2) \rfloor + 4) &= 7 + 3 \lfloor \log(n) - 1 \rfloor \\ &= 4 + 3 \lfloor \log(n) \rfloor. \quad \square \end{aligned}$$

Question 2.5. For each of $n = 2, 3,$ and 4 find two examples of arrays and a number S , one that requires a full $3 \lfloor \log(n) \rfloor + 4$ comparisons and one that requires fewer.

Question 2.6. Suppose that the director's file has 1000 records in it. In the worst case, how many comparisons will it take to find a record with a particular social security number on it if (a) SEQSEARCH is used and (b) BINARYSEARCH is used?

How did we originally find the bound $3 \lfloor \log(n) \rfloor + 4$ of Theorem 2.1? This expression works in the inductive proof, but why? Suppose that $B(n)$ denotes the

maximum number of comparisons made by BINARYSEARCH on an array of n elements. Then in the worst case we perform three comparisons (in steps 3, 5, and 6) and then face a smaller array with $\lfloor n/2 \rfloor$ elements in which to search for S . $B(\lfloor n/2 \rfloor)$ denotes the maximum number of comparisons needed to search this smaller array and so

$$B(n) = 3 + B(\lfloor n/2 \rfloor) \quad \text{and} \quad B(1) = 4. \quad (*)$$

This fact doesn't solve the problem immediately but can lead to a solution as outlined in Exercises 13 to 15. In Chapter 7 we pursue a systematic study of how, given an equation like that of line (*), we can find an expression for the number of comparisons (or other significant operations) performed in the worst case of an algorithm.

From Theorem 2.1 we can get an estimate of the amount of work the director must do each week. If each week one memo arrives for each of the n student employees, then the next result gives an upper bound on the number of comparisons necessary.

Corollary 2.2. BINARYSEARCH requires at most

$$3n\lceil \log(n) \rceil + 4n = O(n \log(n))$$

comparisons to search an ordered array for each of the n files located in it.

This result is an immediate consequence of Theorem 2.1; however, $3n\lceil \log(n) \rceil + 4n$ is really an overestimate. A tighter upper bound on the number of comparisons, but one that is still $O(n \log(n))$, is derived in Exercises 9 to 11. In any case the worst-case behavior of BINARYSEARCH is significantly better than that of SEQSEARCH. Indeed the worst-case performance of BINARYSEARCH is better than the average-case performance of SEQSEARCH. The average-case performance of BINARYSEARCH is analyzed in Exercise 12. In its defense it should be emphasized that SEQSEARCH will work on any set in an array A regardless of whether or not the elements of A form a totally ordered set.

In the next section we use the ideas of binary search to construct a more efficient sorting algorithm.

EXERCISES FOR SECTION 2

1. Let $A = \langle 1, 2, \dots, 7 \rangle$, $B = \langle 2, 4, 6, \dots, 16 \rangle$, and $C = \langle 1, 3, 7, 15, 31, 63 \rangle$. Trace BINARYSEARCH to find (a) $S = 3$ in A , (b) $S = 8$ in A , (c) $S = 6$ in B , (d) $S = 7$ in B , (e) $S = 31$ in C , and (f) $S = 14$ in C .
2. Suppose that the number on the director's memo is less than that on the $\lfloor (n+1)/2 \rfloor$ nd record. What is the index of the next record she consults? Express

this as a function of n . If the number is greater than that on the $\lfloor (n+1)/2 \rfloor$ nd record, what is the index of the next record she consults?

3. Find all values of n for which SEQSEARCH uses fewer comparisons in the worst case than BINARYSEARCH.
4. Find a value of N such that SEQSEARCH uses at least twice as many comparisons in the worst case as BINARYSEARCH. Show that for every $n > N$ SEQSEARCH will always use at least twice as many comparisons in the worst case as BINARYSEARCH.
5. In the worst case, how many subintervals of the form $\langle a_{\text{first}}, \dots, a_{\text{last}} \rangle$ does BINARYSEARCH examine in an array with n entries?
6. Suppose that A is an array containing n words, where each word is a (finite) sequence of letters taken from the English alphabet. Suppose further that your computer can answer the following questions:

given words w and w' , does $w = w'$?
does w precede w' alphabetically?

Write a version of BINARYSEARCH that upon input of A , an array of words listed in alphabetical order, and a word w , searches for w in A .

7. Suppose that we are searching an ordered array of n elements for an element that is in position k (but we don't know that). For what values of k will SEQSEARCH use fewer comparisons than BINARYSEARCH?
8. Modify BINARYSEARCH so that, given an array A with entries in increasing order ($a_1 \leq \dots \leq a_n$) and a number S , it finds all indices i such that $a_i = S$.
9. Let $n = 2^k - 1$. Suppose that payroll memos for n students come into the financial aid office in random order and that records for these n students are arranged by increasing social security number. For each memo BINARYSEARCH is used to locate the appropriate record. At some point, the memo for the $\lfloor (n+1)/2 \rfloor$ nd student arrives and requires only two comparisons to find the correct record. Memos for two other students will require exactly five comparisons.
 - (a) Which numbered students are these?
 - (b) How many memos require exactly eight comparisons to locate their records?
 - (c) What is the next smallest number of comparisons needed and how many students need this many?
 - (d) For each possible value of i , determine the number of memos that require exactly i comparisons.
10. Prove that

$$1 \cdot 2 + 2 \cdot 5 + 4 \cdot 8 + \dots + 2^{i-1}(3i - 1) + \dots + 2^{k-1}(3k - 1) = (3k - 4)2^k + 4.$$

11. Suppose that $n = 2^k - 1$. Then explain why using BINARYSEARCH to search an ordered array for each of n records requires

$$(3k - 4)2^k + 4 = 3n\lfloor \log(n) \rfloor - n + 3\lfloor \log(n) \rfloor + 3 \\ = O(n \log(n))$$

comparisons. Is this bound on the number of comparisons better than that given in Corollary 2.2?

12. Use the results of the preceding exercises to obtain the average number of comparisons used per record in BINARY SEARCH in the case $n = 2^k - 1$. Compare this average with that of SEQSEARCH.
13. Suppose that

$$B(n) = B(\lfloor n/2 \rfloor) + 3 \quad \text{for } n > 1, \quad (*)$$

and

$$B(1) = 4.$$

Determine the value of $B(n)$ for $n = 2, 3, 4, 5, 8,$ and 16 .

14. Suppose that $n = 2^k$. Use (*) repeatedly to determine a formula for $B(n)$. Prove your formula correct by using induction and the equation in (*).
15. Verify that $f(n) = 3\lfloor \log(n) \rfloor + 4$ gives the same values as those obtained for $B(n)$ in Exercise 14. Then prove by induction that $B(n) = f(n)$ satisfies the equation in (*).

6:3 SORTING A FILE

We have seen that searching for one record in an unsorted file with n records in it requires $O(n)$ comparisons in the worst case. This contrasts with a worst case of $O(\log(n))$ comparisons in searching a sorted file. A natural question to ask is whether or not it's better to sort before searching or not. For the moment let's return to the problem of searching the file for each of the n records during every payroll period. If there are t payroll periods and the file remains unsorted, the total number of comparisons required will be $O(tn^2)$. On the other hand, if the director uses SELECTSORT to place the file in order, then the total number of comparisons will be

$$O(n^2) + O(tn \log(n)). \quad (A)$$

If, for example, there were n payroll periods (so $t = n$), then the number of comparisons would be $O(n^3)$ without sorting and $O(n^2 \log(n))$ with sorting. Thus, if the number of payroll periods is large, sorting before searching pays off. Suppose, for contrast, that the number of payroll periods is a small constant. Is it

better to sort before searching or not? If the only sorting algorithm available were SELECTSORT, then both solutions be $O(n^2)$. However, if there were a better sorting algorithm, then one could expect sorting before searching to be faster.

SEQSEARCH requires, on average $(n + 1)/2$ comparisons to position a record correctly within a file containing n records. To sort more economically, we need a way to position a record correctly using fewer comparisons. BINARYSEARCH provides just such a mechanism.

Problem. Given an ordered array of numbers $A = (a_1, a_2, \dots, a_r)$ with $a_1 \leq a_2 \leq \dots \leq a_r$ and a number D , insert D in the ordered list.

We develop the procedure BININSERT that will insert a number D into its correct position in an ordered array. The parameters of the procedure are $(r, a_1, \dots, a_r, a_{r+1})$. We assume that upon calling the procedure the r numbers a_1, \dots, a_r are in order and that a_{r+1} equals D . Upon return a_1, \dots, a_{r+1} should be in order. Within the procedure we repeatedly compare D with the midpoint of a subarray in order to find its correct location. Once D 's correct location is determined, the elements that should follow it are shifted over one space in order to make room for D . We make this algorithm a procedure, since we shall use it within BINARYSORT, which will be our first efficient sorting routine.

Procedure BININSERT($r, a_1, \dots, a_r, a_{r+1}$)

{The initial segment of the procedure finds the correct location for a_{r+1} .}

STEP 1. Set first := 1, last := r

STEP 2. While first \leq last do

 Begin

 STEP 3. Set mid := $\lfloor (\text{first} + \text{last})/2 \rfloor$

 STEP 4. If $a_{r+1} < a_{\text{mid}}$, then set last := mid - 1,
 Else set first := mid + 1

 End {Step 2}

{At this point first equals last + 1, and first gives the correct position for a_{r+1} . The next segment creates a space for and inserts a_{r+1} .}

STEP 5. If first = $r + 1$, then Return. { a_{r+1} 's place is correct.}

STEP 6. Set temp := a_{r+1} {save a_{r+1} }

STEP 7. For $j = r + 1$ down to (first + 1) do

 STEP 8. $a_j := a_{j-1}$

STEP 9. Set $a_{\text{first}} := \text{temp}$

STEP 10. Return.

Example 3.1. Table 6.3 is a trace of the procedure BININSERT given the array $A = (3, 5, 8, 10, 14)$, $r = 5$, and $D = 11$.

Table 6.3

Step No.	first	last	mid	a_{mid}	j	A
3	1	5	3	8		(3,5,8,10,14, 11)
4	4	5	3	8		
3	4	5	4	10		
4	5	5	4	10		
3	5	5	5	14		
4	5	4				
8	5				6	(3, 5,8,10,14, 14)
9	5				6	(3, 5,8,10,11, 14)

Question 3.1. Trace BININSERT if $A = (2,5,7,9, 13,15, 19)$ and $D =$ (a) 1, (b) 4, (c) 14, and (d) 23.

Notice the similarity between BINARYSEARCH and BININSERT. The test for equality has been eliminated because if $a_r + 1 = a_{mid}$, this procedure correctly inserts $a_r + 1$ in position $mid + 1$ or higher. Exercise 12 outlines a proof that BININSERT works correctly.

Question 3.2. If $A = (2,5,7,9,13,15, 19)$, trace BINARYSEARCH and BININSERT with $S = D = 16$. Compare the two algorithms.

Before discussing the complexity of BININSERT, we use this procedure to develop an algorithm to totally order an array.

Problem. Given an array of n numbers (a_1, a_2, \dots, a_n) , place them in increasing order.

Algorithm BINARYSORT

- STEP 1. Input n and an array (a_1, \dots, a_n)
 STEP 2. For $m = 2$ to n do {insert m th item}
 STEP 3. Call BININSERT $((m - 1), a_1, \dots, a_m)$
 STEP 4. stop.

Question 3.3. Given the array $(13, 23,17,19,18, 28)$ trace out the algorithm BINARYSORT.

Once we determine the complexity of the procedure BININSERT, the complexity of algorithm BINARYSORT will be easy to analyze, since BININSERT is used $n - 1$ times in BINARYSORT. The steps in BININSERT are either assignments or comparisons. We count the latter.

Theorem 3.1. BININSERT requires at most $2\lfloor \log(r) \rfloor + 4$ comparisons to insert the $(r + 1)$ st term into an already sorted list of r items.

Proof. The only steps containing comparisons are steps 2, 4, and 5, and each of these executes exactly one comparison. We proceed by induction. If $r = 1$, then after the first execution of step 4, either $\text{first} = 1$ and $\text{last} = 0$ or $\text{first} = 2$ and $\text{last} = 1$, depending on whether a_2 is less than a_1 or not. Step 2 is repeated to check this. Step 5 is required to rearrange the array. Thus four comparisons are used in total.

The induction hypothesis will be that for $t < r$ BININSERT requires at most $2\lfloor \log(t) \rfloor + 4$ comparisons to insert the $(t + 1)$ st item into any already sorted list with t items.

We suppose that A is an ordered array with r elements and $a_{r+1} = D$ is to be inserted. It takes two comparisons to execute through step 4 the first time. After the first execution of step 4, if $a_{r+1} < a_{\text{mid}}$ then last is assigned the value $\text{mid} - 1$. Thus we restrict our attention to $(a_1, \dots, a_{\text{mid}-1}, a_{r+1})$. There are $\text{mid} - 1$ ordered values in this array. Now

$$\text{mid} - 1 = \left\lfloor \frac{1+r}{2} \right\rfloor - 1 = \left\lfloor \frac{r-1}{2} \right\rfloor < \frac{r}{2}.$$

After the first execution of step 4 if $a_{r+1} \geq a_{\text{mid}}$, then first is assigned the value $\text{mid} + 1$. Thus we restrict our attention to $(a_{\text{mid}+1}, \dots, a_r, a_{r+1})$. The number of elements in this smaller ordered array is $(r - (\text{mid} + 1) + 1) = r - \text{mid}$. Now if $r = 2j$,

$$\begin{aligned} r - \text{mid} &= r - \left\lfloor \frac{1+r}{2} \right\rfloor \\ &= 2j - \left\lfloor \frac{1+2j}{2} \right\rfloor = 2j - j = j = \frac{r}{2}. \end{aligned}$$

On the other hand, if $r = 2j + 1$,

$$\begin{aligned} r - \text{mid} &= 2j + 1 - \left\lfloor \frac{1+2j+1}{2} \right\rfloor \\ &= 2j + 1 - (j + 1) = j < \frac{r}{2}. \end{aligned}$$

Thus in either case the smaller ordered array has no more than $r/2$ entries. By the inductive hypothesis we can insert D into the new array using at most $2\lfloor \log(r/2) \rfloor + 4$ comparisons. Thus the total number of comparisons required will

be at most

$$\begin{aligned} 2 + (2\lfloor \log(r/2) \rfloor + 4) &= 2\lfloor \log(r) - 1 \rfloor + 6 \\ &= 2\lfloor \log(r) \rfloor + 4. \end{aligned} \quad \square$$

Question 3.4. For each of the examples in Question 3.1 count the number of comparisons and verify that these are each no more than $2\lfloor \log(7) \rfloor + 4$.

The formula $2\lfloor \log(n) \rfloor + 4$ in the preceding complexity analysis appears out of the blue. That BININSERT and BINARYSEARCH have similar complexity analyses is not surprising. To motivate the particular formula we obtain, we examine the proof of Theorem 3.1. If $C(n)$ denotes the maximum number of comparisons made when BININSERT inserts a number into a sorted array of length n , then

$$c(n) = C(\lfloor n/2 \rfloor) + 2,$$

since two comparisons are performed, and then the algorithm proceeds to work on an array containing at most $\lfloor n/2 \rfloor$ entries. This equation for $C(n)$ is like that of line (*) of Section 6.2 and can be used to derive the formula $C(n) = 2\lfloor \log(n) \rfloor + 4$. This derivation will be discussed in depth in Chapter 7.

Theorem 3.2. The number of comparisons required by BINARYSORT to order an array of n numbers is $O(n \log(n))$.

Proof. The only comparisons in BINARYSORT are performed within the BININSERT procedure. BININSERT is called $n - 1$ times. The number of comparisons in each call is at most $2\lfloor \log(n - 1) \rfloor + 4$. Thus the total number of comparisons will be no more than

$$\begin{aligned} (n - 1)(2\lfloor \log(n - 1) \rfloor + 4) &< n \{2(\log(n)) + 4\} \\ &\leq 6n \log(n). \end{aligned} \quad \square$$

BINARYSORT is thus considerably more efficient than SELECTSORT, a $O(n^2)$ algorithm.

Question 3.5. Count the number of comparisons made in Question 3.3 and compare this number with $(n - 1)(2\lfloor \log(n - 1) \rfloor + 4)$ for $n = 6$.

It is instructive to contrast the analyses presented in Theorems 3.1 and 3.2. We showed that BININSERT required at most $2\lfloor \log(n) \rfloor + 4$ comparisons to insert the $(n + 1)$ st item into an already sorted array. In Exercises 9 and 10 you

will see that this bound is sharp. We mean that there are problem instances where $2\lfloor \log(n) \rfloor + 4$ comparisons are, in fact, required. Thus there can be **no** upper bound for the number of comparisons that is always better than the one given in Theorem 3.1.

Notice that our analysis of BINARYSORT was not so sharp. In particular, we assumed that each call to BININSERT needed the full $2\lfloor \log(n-1) \rfloor + 4$ comparisons whereas we really need only $2\lfloor \log(1) \rfloor + 4$ comparisons for the first insertion, $2\lfloor \log(2) \rfloor + 4$ comparisons for the second, and, in general, $2\lfloor \log(i) \rfloor + 4$ comparisons for the i th insertion. Thus the total number of comparisons we perform is at most

$$\begin{aligned} & (2\lfloor \log(1) \rfloor + 4) + \cdots + (2\lfloor \log(n-1) \rfloor + 4) \\ & \leq (2\log(1) + 4) + \cdots + (2\log(n-1) + 4) \\ & = 4(n-1) + 2\{\log(1) + \cdots + \log(n-1)\} \\ & = 4(n-1) + 2\log((n-1)!). \end{aligned} \tag{B}$$

Exercise 13 asks you to use equation (B) to provide a smaller upper bound than the one obtained so far for BINARYSORT. However, no analysis of the complexity of BINARYSORT can demonstrate that it is more efficient than $O(n \log(n))$. The goal of Section 5 is to show that SELECTSORT, BINARYSORT, and every sorting method that uses comparisons must perform at least a constant times $n \log(n)$ comparisons in the worst case. Before we get to that, we shall see in the next section that trees provide an illustrative model of these searching and sorting algorithms.

What effect does BINARYSORT have on the employment director's work load, as presented in the first paragraph of this section? If she first sorts the employment file using BINARYSORT, using $O(n \log(n))$ comparisons, and then during t time periods processes information using BINARYSEARCH with $O(tn \log(n))$ comparisons, then the total number of comparisons is

$$O(n \log(n)) + O(tn \log(n)) = O(tn \log(n)). \tag{C}$$

Comparing the results of (C) with those of (A), we see that the latter process is at least as efficient as the former and for some values of t is more efficient.

EXERCISES FOR SECTION 3

1. Trace BININSERT on the following data:

(a) $A = (1,2,3)$, $D = 2.5$.

(b) $A = (1,2,3)$, $D = 0$.

(c) $A = \langle 1,2,4 \rangle$, $D = 2$.

(d) $A = (2,3,5,7, 11,13,17, 19)$, $D = 12$.

(e) $A = (2,4,6,8, 10)$, $D = 5$.

2. Count the number of comparisons made in each part of Exercise 1. Compare this number with $2\lfloor \log(r) \rfloor + 4$ for the appropriate values of r .
3. Here is another algorithm to search for D in an array A :

```

STEP 0. Input  $A = \langle a_1, a_2, \dots, a_r \rangle$ , set  $a_{r+1} := D$ 
STEP 1. Set first := 1, last :=  $r$ 
STEP 2. While first < last do
    Begin
        STEP 3. Set mid :=  $\lfloor (\text{first} + \text{last})/2 \rfloor$ 
        STEP 4. If  $a_{r+1} \leq a_{\text{mid}}$ , then set last := mid
                Else set first := mid + 1
    End
STEP 5. If  $a_{\text{first}} = a_{r+1}$ , then output "found  $D$  at location first" and stop.
        Else output " $D$  is not in  $A$ " and stop.

```

Run this algorithm and BINARYSEARCH on $A = \langle 1, 2, 3, 4, 5, 6, 7 \rangle$ with $D = 1.5, 2$, and 2.5 .

4. Compare the algorithm BINARYSEARCH and that given in the preceding exercise. Determine which one performs fewer comparisons.
5. Suppose that the employment director uses SEQSEARCH on an unsorted file of n records to register n students' payroll data during t time periods, making $f(n, t)$ comparisons as described in the first paragraph of this section. Let $g(n, t)$ denote the number of comparisons made if the file is first sorted using SELECTSORT and then the same recordings are made using BINARYSEARCH [see line (A) in text]. Find the smallest value of t such that $g(n, t) < f(n, t)$.
6. Let $g(n, t)$ be as defined in the preceding problem and let $h(n, t)$ be the number of comparisons made if the file is first sorted using BINARYSORT and then the memos are recorded for n students on n records in t time periods using BINARYSEARCH. Compare $g(n, t)$ and $h(n, t)$.
7. Run BINARYSORT on each of the following (a) $A = (1, 2, 3)$, (b) $A = (2, 1, 3)$, (c) $A = (3, 1, 2)$, and (d) $A = \langle 3, 2, 1 \rangle$.
8. Count the number of comparisons made in the preceding exercise and compare this number with $(n-1)(2\lfloor \log(n-1) \rfloor + 4)$ for the appropriate value of n .
9. For an n of your choice find an example of an array A of n numbers and a number D on which BININSERT performs exactly $2\lfloor \log(n) \rfloor + 4$ comparisons.
10. Let $n = 2^k$ with k an arbitrary positive number. Describe an array A of n numbers and another number D on which BININSERT performs exactly $2\lfloor \log(n) \rfloor + 4 = 2k + 4$ comparisons.

11. Explain why BINARYSORT will always perform fewer than $6n \log(n)$ comparisons when sorting an array of length n . Will BINARYSORT perform fewer than $(n-1)(2\lceil \log(n-1) \rceil + 4)$ comparisons on any or all arrays of length n ?
12. Prove that BININSERT works correctly by proving each of the following statements.
 - (a) While $\text{first} \leq \text{last}$, a_{r+1} should be stored in one of the entries a_{first} , $a_{\text{first}+1}, \dots$, or $a_{\text{last}+1}$. In particular, check that this is so when $a_{r+1} = a_{\text{mid}}$.
 - (b) Eventually either last equals first or $\text{first}+1$.
 - (c) If last equals first or $\text{first}+1$, then BININSERT places a_{r+1} in the correct position.
13. Stirling's formula, discussed in Chapter 3, implies that

$$n! = O\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right).$$

Use this result together with equation (B) to derive an upper bound on the number of comparisons made in BINARYSORT. How does this upper bound compare with the upper bound derived in the text?

14. Suppose that A is an array of n elements that is already sorted (but we may not know that in advance). Which algorithm works faster on A , SELECT-SORT or BINARYSORT? Explain.

6:4 SEARCH TREES

Suppose that we search an ordered array $A = \langle a_1, a_2, \dots, a_7 \rangle$ for a particular object S . BINARYSEARCH would have us first compare S with a_4 . There are three possible outcomes of such a comparison. If $S = a_4$, we're done. If $S < a_4$ and S is in A , then it must be one of a_1, a_2 , or a_3 . Finally, if $S > a_4$ and S is in A , then it must be one of a_5, a_6 , or a_7 . In this section we show how to use a tree structure to illuminate these logical alternatives.

Recall from Chapter 5 that we can think of a graph as a set of points in the plane and a set of line segments or arcs joining pairs of these points. A graph that is both connected and acyclic is called a tree.

Here is how BINARYSEARCH as applied to the seven-element set $A = \langle a_1, a_2, \dots, a_7 \rangle$ can be modeled by a path within a tree of seven vertices. Begin with a single vertex labeled a_4 . Think of two edges coming out of a_4 , one labeled by " $<$ " and the other labeled by " $>$ ". (See Figure 6.1.) The " $<$ " edge joins a_4 with a_2 and the " $>$ " edge joins a_4 with a_6 . In terms of BINARYSEARCH if S equals a_4 , we stay at the vertex labeled a_4 and we are done. If $S < a_4$ we proceed along the

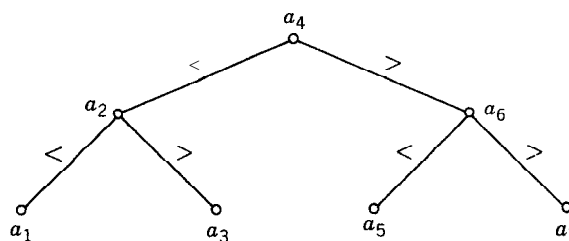


Figure 6.1

edge labeled $<$ to the vertex labeled a_2 . If $S > a_4$ we proceed along the edge labeled $>$ to the vertex labeled a_6 . The vertices a_2 and a_6 each have two additional edges coming out of them labeled with $<$ and $>$. The new edge from a_2 labeled $<$ terminates at a_1 while the new edge from a_2 labeled $>$ terminates at a_3 . Similarly, the new edges from a_6 terminate at a_5 and a_7 . For example, if $S = a_5$, BINARYSEARCH would examine a_4 , followed by a_6 and then a_5 . If S is not present in A , we should also perform comparisons, for example, with a_2 , a_6 , and a_5 and then deduce that S was not in A . In all cases these comparisons correspond to a path within the so-called search tree shown in Figure 6.1.

Question 4.1. Draw a search tree to illustrate a binary search of an array of 15 elements.

Recall that within a graph the degree of a vertex is the number of edges incident with that vertex. In a tree or forest a vertex of degree 1 is called a leaf.

Definition. A tree is called **binary** if

- (1) it possesses a distinguished vertex called the **root** whose degree is either 2 or 0, and
- (2) every vertex of the tree other than the root has degree either 3 or 1.

Note that the tree in Figure 6.1 is binary. It is customary to draw a binary tree “upside down” with the root at the top, as in Figure 6.1. From the root (if its degree is not 0) there is a left edge down and a right edge down. Similarly, every other vertex that is not a leaf has a left and a right edge down. One of the nice properties that binary trees with three or more vertices have is that if the root and its incident edges of a binary tree are erased, then two smaller binary trees are formed. These are called the left and right **subtrees** of the original tree.

Example 4.1. Figure 6.2 exhibits a binary tree with five vertices.

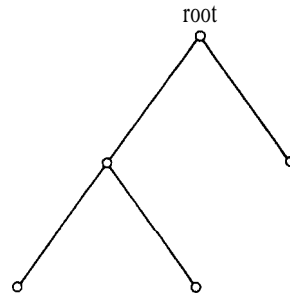


Figure 6.2

Question 4.2. (a) Draw all binary trees with fewer than eight vertices; (b) draw all binary trees with two, three or four leaves.

Question 4.3. Draw the left and right subtrees that are formed when the root and its two incident edges are deleted from (a) the tree in Figure 6.1; (b) the tree in Figure 6.2; and (c) every tree with two or three leaves (see Question 4.2).

Definition. In any tree with a designated root the depth or level of a vertex x is the number of edges in the unique path from x to the root. The depth of the tree is the maximum depth of any vertex in the tree. Alternatively, it is the length of a longest path from the root.

In the tree shown in Figure 6.1, a_4 , the root, is at level 0; a_2 and a_6 are at level 1; and a_1, a_3, a_5 , and a_7 are all at level 2. Thus the tree has depth 2.

Question 4.4. If T is a binary tree of depth $d > 0$ and T' is the left subtree of T , what can you say about the depth of T' ?

Theorem 4.1. A binary tree has at most 2^d vertices at depth d .

Proof. The proof is by induction on d . The root is the only vertex at level 0. By definition there are either two or zero vertices adjacent to the root, and these vertices are at level 1. We assume that there are no more than 2^k vertices at level k in a binary tree. Consider level $k + 1$. Every vertex at this level must be adjacent to exactly one vertex at level k by the definition of tree (see Exercise 1). Since each vertex at level k has degree 1 or 3, it is adjacent to either zero or two vertices at level $k + 1$. Thus the number of vertices at level $k + 1$ can be no more than twice the number of vertices at level k . If N_k denotes the number of vertices at level k , we have

$$N_{k+1} \leq 2N_k \leq 2(2^k) = 2^{k+1}. \quad \square$$

Corollary 4.2. A binary tree of depth d contains at most $2^{d+1} - 1$ vertices.

Proof. A binary tree of depth d has vertices at levels $0, 1, \dots, d$. By the preceding theorem there are at most 2^k vertices at level k . Thus the total number of vertices in the tree is at most

$$\begin{aligned} 1 + 2 + 4 + \dots + 2^k + \dots + 2^d &= \frac{1 - 2^{d+1}}{1 - 2} && \text{by Question 2.3.3} \\ &= 2^{d+1} - 1 && \square \end{aligned}$$

A binary tree of depth d with $2^{d+1} - 1$ vertices is called a full binary tree.

Question 4.5. Determine the depth and the number of vertices in the smallest full binary tree that has n or more leaves.

Now we specify the connection between binary trees and our problem of searching a sorted array. Suppose that the array A contains $n = 2^k - 1$ elements in order. Then the corresponding binary tree will be a full binary tree of depth $k - 1$ with n vertices. In BINARYSEARCH the first element of the array A with which we compare S is the element a_{mid} . Note that $\text{mid} = 2^{k-1}$. The element a_{mid} will label the root of the binary tree that we are about to search. If we find that $S = a_{\text{mid}}$ (in step 5), then the algorithm stops. We can similarly stop searching the tree. If $S < a_{\text{mid}}$, we set

$$\text{last} := \text{mid} - 1 = 2^{k-1} - 1$$

and

$$\text{mid} := \left\lfloor \frac{(\text{first} + \text{last})}{2} \right\rfloor = 2^{k-2}.$$

This corresponds with traversing the edge from the root of the binary tree down to the root of the left subtree; this vertex is labeled with the new $a_{\text{mid}} = a_{2^{k-2}}$. Similarly, if $S > a_{\text{mid}}$, we traverse an edge to the right subtree. If we have not found S , we repeat this process. Each time we examine a new a_{mid} it will be the root of a subtree of the original binary tree. Each such subtree will contain $2^j - 1$ vertices for some j . In the end either we find S and terminate our path down from the root of the tree or we reach a leaf without finding S and stop. The number of comparisons made in BINARYSEARCH is the same as the number of vertices visited on the corresponding path in the tree.

More generally, if A contains n elements where

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

then

$$k - 1 < \log(n) < k,$$

and

$$k - 1 = \lfloor \log(n) \rfloor.$$

Set

$$n' = 2^k - 1.$$

We model the search of A by a search of the full binary tree of depth $k - 1$ containing n' vertices, labeled as before. If $n' > n$, some of the labels of vertices, namely $a_{n+1}, \dots, a_{n'}$, do not correspond with array elements.

Question 4.6. Compute n' if $n = 15, 26$, and 31 . Show that, in general, $n' \geq n$.

Question 4.7. Draw and label the tree that corresponds with a binary search of a 23-element ordered array.

One advantage of the binary tree model of BINARYSEARCH is that it supports a simple complexity analysis. Suppose that we are searching an ordered array of n elements. If $2^{d-1} < n < 2^d$, then the elements of the array correspond with some of the vertices of a full binary tree of depth $d - 1$. Comparing S with array elements corresponds with visiting vertices in the tree. Since each time we traverse an edge to the root of a new subtree, the depth of the visited vertex increases, we shall in the worst case visit vertices at depth $0, 1, \dots$ and $(d - 1)$. Correspondingly, we need to compare S with no more than d elements of the array. Since a search tree with n vertices has depth $\lfloor \log(n) \rfloor$, we can determine whether S is present in an ordered array of n elements by examining no more than $\lfloor \log(n) \rfloor$ elements of the array. If it takes just a constant number of steps for each such examination, then it is immediate that BINARYSEARCH is $O(\log(n))$. In contrast in Exercise 15 we explore how SEQSEARCH can be modeled by searching a graph that is just a path.

It is possible to think of a binary tree model of the first four steps of BININSERT in much the same way. Recall that these steps determine the location of the next element to be inserted. Continuing the model is awkward because the insertion of a single element can cause a radical change in the binary tree. Exercise 13 illustrates this.

Although it is difficult to use trees to model BINARYSORT, there is an elegant sorting method called TREESORT that is based on binary trees. Suppose that we want to sort $A = \langle a_1, \dots, a_n \rangle$, where the entries of A are distinct. (See Exercise 20 for the case of repeated elements.)

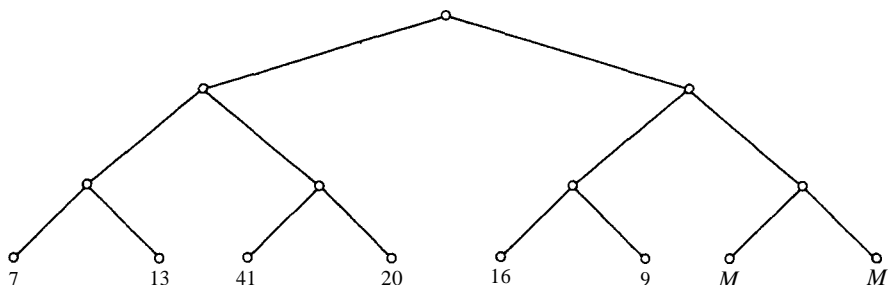


Figure 6.3

Algorithm TREESORT

STEP 1. Set $k = \lceil \log(n) \rceil$ and construct the full binary tree of depth k . {As you saw in Question 4.5, this tree has at least n leaves.} Assign each element in the array to a leaf. Pick a number M that is greater than any element in the array and assign M to every blank leaf.

Example 4.2. Given $A = (7, 13, 41, 20, 16, 9)$, $k = \lceil \log(6) \rceil = 3$. Figure 6.3 exhibits the full binary tree of depth 3 with its leaves labeled.

STEP 2. For $j = k - 1$ down to 0 do
 Assign to each vertex at level j the minimum of the two values assigned to its neighbors at level $j + 1$.

Example 4.2 (continued). We show in Figure 6.4 the full binary tree after step 2. (At this stage the minimum value in the array is assigned to the root of the binary tree.)

STEP 3. Set $b_1 :=$ value assigned to the root
 STEP 4. For $i = 2$ tondo
 Begin
 STEP 5. Erase every occurrence of b_{i-1} from nodes of the tree.
 STEP 6. Assign M to the leaf that originally was labeled with b_{i-1}
 STEP 7. For $j = k - 1$ down to 0 do
 Assign to the vertex at level j that used to be labeled b_{i-1} the minimum of the two values assigned to its neighbors at level $j + 1$
 STEP 8. Set $b_i :=$ value assigned to the root
 End
 STEP 9. stop.

Example 4.2 (continued again). We exhibit in Figure 6.5 the labeled tree after the first execution of step 7; $b_1 = 7$.

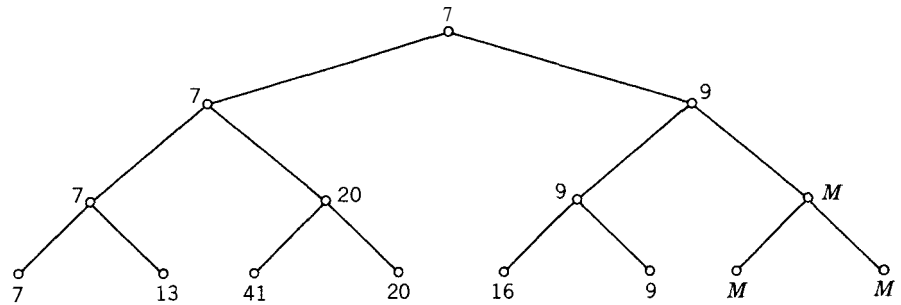


Figure 6.4

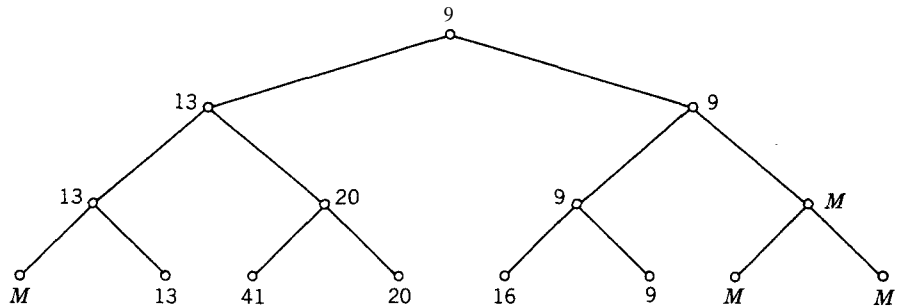


Figure 6.5

After the execution of step 7 the i th smallest value in the array is assigned to the root and thus in step 8 is assigned to b_i .

Question 4.8. Complete this execution of TREESORT.

Exercises 17 to 19 ask you to show that TREESORT is a $O(n \log(n))$ algorithm.

Binary trees are useful models for many topics in combinatorics and computer science, for example, see Exercises 5 and 6. We shall use binary trees again in the following sections and in Chapter 8.

EXERCISES FOR SECTION 4

1. In the proof of Theorem 4.1 we claimed that every vertex at level $k + 1$ is adjacent to exactly one vertex at level k . Why is this so?
2. If x is a vertex in a rooted tree, let $l(x)$ denote the level of x . Show that if v is adjacent to u , then $l(v) = l(u) + 1$ or $l(v) = l(u) - 1$. Give a proof or a counterexample to the converse of this statement.

3. If T is a binary tree of depth d , what is the smallest number of vertices that T might have at level k (for $k = 1, \dots, d$)?
4. What is the smallest number of vertices that a binary tree of depth d might have?
5. For what integers n is there no binary tree with exactly n vertices? For what integers n is there no binary tree with exactly n leaves? Prove your answers by induction.
6. We can also represent the subsets of a set (a_1, a_2, \dots, a_n) with a full binary tree. Suppose that we label the root of the tree with \emptyset , the empty set. Then the left subtree will correspond with subsets not containing a_1 and the right subtree subsets containing a_1 . Similarly, the left subtree within the left subtree will correspond with subsets containing neither a_1 nor a_2 , whereas its right subtree will correspond with subsets containing a_2 but not a_1 . Each node can be labeled with a subset, representing the choices of elements made along the path from the root to that node. Using this idea, construct the binary tree associated with all subsets of a 3-set and of a 4-set.
7. In a full binary tree there are 2^k vertices at level k . Find a correspondence between the subsets of a k -set and the vertices at the k th level of a full binary tree.
8. Prove Theorem 4.1 by “erasing the root.”
9. Prove Corollary 4.2 by “erasing the root.”
10. Suppose that $n = 2^k - 1$ and consider a full binary tree with vertices labeled with the elements of an array $A = \langle a_1, \dots, a_n \rangle$ corresponding with BINARY-SEARCH. Which elements label vertices at depth 1? At depth 2? What are the labels of the leaves?
11. Repeat Exercise 10 in the case of arbitrary n . How can you tell from i and j if a_i and a_j label vertices at the same level?
12. Trace out the path corresponding to BINARYSEARCH when this algorithm is applied to the following arrays, modeled by binary trees, and elements S :
 - (a) A as in Figure 6.1, $S = a_1$.
 - (b) A as in Figure 6.1, $a_3 < S < a_4$.
 - (c) A as in Question 4.7, $S = a_{16}$.
 - (d) A as in Question 4.7, $S = a_{23}$.
 - (e) A as in Question 4.7, $a_{16} < S < a_{17}$.
13. Suppose that you want to insert $D = 31$ into the sorted array $A = (3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36)$. Construct a binary tree to model the sorted array A both before and after the insertion of D . In how many locations do these two trees differ?
14. If T is a binary tree with q leaves, how many vertices of degree 3 does T have?

15. Explain how the algorithm SEQSEARCH can be modeled by traversing a graph that is a path.
16. Run TREESORT on the following arrays, showing the binary tree and its node values at the end of each execution of steps 3 and 8: **(a)** $A = \langle 1, 2, 3 \rangle$; **(b)** $A = (2, 1, 3)$; and **(c)** $A = (1, 5, 2, 6, 3, 4)$.
17. How many comparisons are performed in step 2 of TREESORT?
18. In TREESORT how may b_{i-1} 's get erased the i th time through the loop? How many comparisons will you need to relabel the tree?
19. Show that TREESORT is a $O(n \log(n))$ algorithm.
20. Rewrite TREESORT so that it sorts arrays with repeated entries.

6:5 LOWER BOUNDS ON SORTING

A new employment director wants to improve the efficiency of the student employment office. After conversations with the previous director, the new director is convinced that she should sort her payroll drawer at the beginning of the year. She learns that the previous director originally used the quadratic algorithm SELECTSORT and then switched to the $n \log(n)$ algorithm BINARYSORT, but the new director does not want to keep switching algorithms each year when the local algorithm experts come up with new and faster (and possibly more complex) sorting algorithms. She decides she would like once and for all to find a fastest sorting algorithm and guesses that there must be a linear algorithm, one that runs in $O(n)$ -time on an array of n elements. So she calls her friends taking the computer science course on algorithms and asks for such a linear-time sorting algorithm.

The algorithm students report that they haven't learned about such an algorithm yet, but maybe they will later in the semester. In the meantime they suggest that the director might like to try Treesort, Shakersort, or Mergesort. These are all $n \log(n)$ algorithms. The director rejects these offers. She is trying to run an efficient office and is not interested in becoming an algorithmic specialist herself. However, she decides that she will, on her own, search for a linear-time sorting algorithm or else prove that there is no such algorithm.

The new director has studied some discrete mathematics and begins with small examples of arrays. If she has an array like $\langle a_1, a_2, a_3 \rangle$, then in how many different orders might the array appear? For example, the array might be "in order" so that $a_1 \leq a_2 \leq a_3$ or "out of order" with $a_1 \leq a_3 \leq a_2$ or $a_2 \leq a_3 \leq a_1$, and so on. How many comparisons must be made to sort these elements into increasing order? The element a_1 can be compared with a_2 and with a_3 , and the elements a_2 and a_3 can be compared with each other. Are all these comparisons necessary?

Question 5.1. For $n = 3, 4,$ and 5 , given an array of n distinct items $A = \langle a_1, a_2, \dots, a_n \rangle$ decide in how many different possible orderings these elements

might appear. Then determine the total number of pairwise comparisons that can be made among the members of the set.

Proposition 5.1. There are at most $n!$ different possible orderings of an array of n elements. In such an array there are $n(n-1)/2$ distinct pairs of items that might be compared.

Proof. If the array entries are distinct, there are the same number of orderings as permutations of an n -set. If the entries are not distinct, the number of different orderings is less than $n!$, since some permutations produce the same ordering. (See also Exercises 2 and 3.) In either case the number of possible comparisons between pairs is the same as the number of 2-subsets of an n -set or (equivalently) the number of edges in an n -clique. \square

Proposition 5.1 tells us (and the employment director) that if we make all possible comparisons, we have a $O(n^2)$ algorithm, an algorithm as slow as SELECTSORT. We know we can use $cn \log(n)$ comparisons for some constant c , but can we use even fewer than this?

Any sorting algorithm that uses comparisons contains a sequence of comparisons, say C_1, C_2, \dots, C_k . Regardless of the particular sorting algorithm used, what can we say about the value of k , the number of comparisons, in the worst case? We model the problem with a binary search tree.

Example 5.1. Given three distinct objects, say a_1, a_2 , and a_3 , we make a binary search tree labeled with possible comparisons and possible outcomes. Suppose that we begin by comparing a_1 and a_2 . There are two possible outcomes, either $a_1 \leq a_2$ or $a_1 > a_2$. We label the root of the binary tree with $(a_1 : a_2)$ for this comparison; the edge from the root to the left subtree is labeled “ \leq ” to denote the first possible outcome and the edge to the right subtree is labeled “ $>$ ” for the second outcome. See Figure 6.6. Suppose that we next compare a_1 with a_3 and label the two nodes on level 1 with $(a_1 : a_3)$ and their left edges with “ \leq ” and their right edges with “ $>$ ”. Notice that in two of the four possibilities we know the correct ordering and have written that in as a leaf of the tree, but in the remaining two cases we need to make the additional comparison of a_2 with a_3 .

The results of Example 5.1 show that at least with this order of comparisons three comparisons are needed in the worst case.

Question 5.2. As in Example 5.1 construct and label a binary search tree when comparisons are made in the following order:

- (a) a_1 with a_2 , a_2 with a_3 , a_1 with a_3 .
- (b) a_1 with a_3 , a_1 with a_2 , a_2 with a_3 .

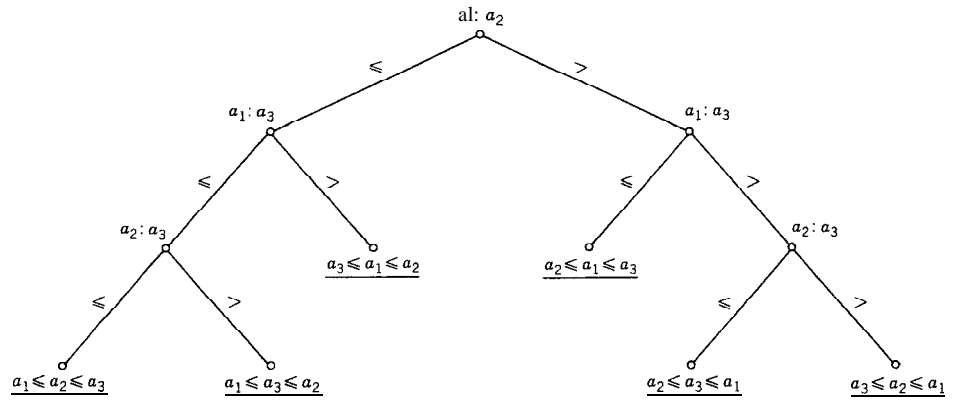


Figure 6.6

The results of Example 5.1 and Question 5.2 convince the employment director that three comparisons are needed to sort a three-element array. This finding is inconclusive from the complexity point of view, since all possible comparisons are needed in the worst case. On the other hand, three comparisons for a three-element array might indicate the possibility of a linear-time algorithm.

Example 5.2. Suppose that we sort $A = \langle a_1, a_2, a_3, a_4 \rangle$. Here is part of the search tree (Figure 6.7). Notice that once it is known that $a_3 < a_1 \leq a_2$, then it is impos-

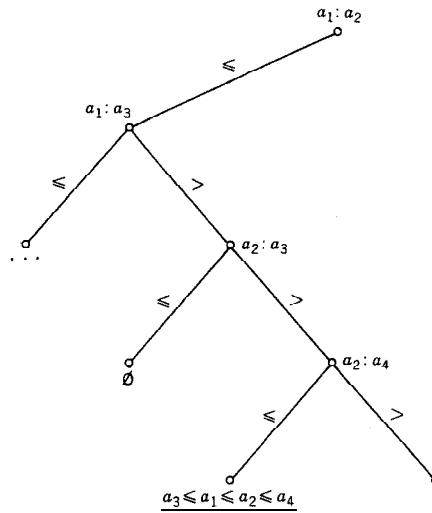


Figure 6.7

sible to have $a_2 \leq a_3$. The path corresponding to this impossibility terminates with a leaf labeled with the empty set.

The director also realizes that she can analyze any sorting algorithm using the binary tree structure. If any algorithm makes comparisons C_1, C_2, \dots, C_k in that order, then we represent this algorithm as a binary search tree of depth k . The root is labeled C_1 , the two nodes at depth 1 are labeled C_2 , and, in general, the nonleaf nodes at depth i are labeled C_{i+1} for $i = 0, \dots, k-1$. The leaves are each labeled with either the empty set or with one of the $n!$ possible orderings of the array. The label on a leaf at the end of a path from the root is the ordering specified by the series of \leq and $>$ s on that path; as in Example 5.2 there may be no such ordering. If k comparisons are made in the worst case, then the binary tree representing these comparisons has depth k . By Theorem 4.1 there are at most 2^k leaves in such a tree. Thus we must have

$$2^k \geq n!, \quad \text{or} \quad k \geq \log(n!). \quad \square$$

We have proved the following theorem.

Theorem 5.2. It requires at least $\log(n!)$ comparisons to sort an n -element set in the worst case.

Another approach to the proof of this theorem, using a game and binary numbers, is given in Exercises 16 and 17.

The director realizes now that an algorithm that performs sorting by comparisons must in the worst case do at least $\log(n!)$ comparisons. Her next goal is to determine the size of $\log(n!)$ or at least a lower bound on $\log(n!)$. The next arithmetic lemma will lead to a lower bound on $\log(n!)$.

Lemma 5.3. If $n \geq i \geq 1$, then $i(n+1-i) \geq n$.

Proof. Since $n \geq i$ and $(i-1) \geq 0$, we have

$$(i-1)n \geq (i-1)i.$$

Adding n to both sides and subtracting $(i-1)i$ yields

$$in - i(i-1) \geq n,$$

which simplifies to

$$i(n-i+1) \geq n. \quad \square$$

We now use the lemma to estimate $\log (n!)$.

$$\begin{aligned}\log (n!) &= \log (n(n-1) \cdots 3 \cdot 2 \cdot 1) \\ &= \log \{[(n)1][(n-1)2][(n-2)3] \cdots [(n-i+1)i] \cdots \cdot\}\end{aligned}$$

by regrouping factors

$$= \log [(n)1] + \log [(n-1)2] + \cdots + \log [(n-i+1)i] + \cdots \quad (\text{A})$$

by the additive property of logs

$$\geq \log (n) + \log (n) + \cdots + \log (n) + \cdots \quad (\text{B})$$

by Lemma 5.3.

Note that if n is even, the sum in (A) ends with $\log [(n/2 + 1)(n/2)]$. Hence there are $n/2$ terms and so (B) equals $(n/2) \log(n)$. Thus

$$\log (n!) \geq \binom{n}{2} \log(n).$$

(For the case of n odd, see Exercise 9.)

Corollary 5.4. If $S(n)$ is defined to be the number of comparisons needed in the worst case to sort a file with n items using comparisons, then

$$S(n) \geq \log (n!) \quad \text{and} \quad n \log (n) = O(S(n)).$$

Corollary 5.4 is often referred to as the information theoretic lower bound on sorting. What it means is that there cannot be a sorting method based on pairwise comparisons whose complexity is of order less than $n \log (n)$. In other words, every such sorting algorithm will be big oh of some function in the functional hierarchy (as developed in Chapter 2) that is at least as big as $n \log (n)$. So far, we have seen three sorting procedures, one that is $O(n^2)$ and two that are $O(n \log(n))$.

Example 5.3. Suppose that we want to sort $A = (a_1, a_2, a_3, a_4)$. As we have already seen, three elements in an array can be sorted with three or fewer comparisons. Suppose that we find that $a_1 \leq a_2 \leq a_3$ and we want to find the position of a_4 . We could compare a_4 with each of the first three to find its position or we could use the idea of BINARYSORT. Then we would compare a_4 with a_2 . If $a_4 \leq a_2$, we compare a_4 with a_1 . If $a_4 > a_2$, then we compare a_4 with a_3 and so determine the final order. In total, five comparisons have been made in the worst case. Since $\lceil \log(4!) \rceil = 5$, Corollary 5.4 tells us that a file with four items cannot be sorted using fewer than five comparisons.

Question 5.3. Draw the binary search tree for a four-element array when the comparisons are made as suggested in Example 5.3. Count the number of comparisons in the worst case. (Note that to parallel Example 5.3 we allow different nodes on the same level of the binary tree to receive different comparison labels.)

Notice that we talk repeatedly about “sorting by comparisons.” How else might a sorting algorithm proceed? In fact, in some special cases comparisons are not necessary; these ideas are explored in Section 8 and in Supplementary Exercises 29 to 32. For example, the algorithm BUCKETSORT is a linear-time sorting algorithm but is of limited applicability because of its excessive storage requirements.

The director is now convinced that there is no algorithm that uses comparisons and is essentially better than BINARYSORT, and so she decides to stick with this $n \log(n)$ algorithm. However, due to increased tuition and decreased financial aid the number of student employees doubles and then grows to triple the original number of students.

Question 5.4. Suppose that the number of student employees increases from 600 to 1200. If the employment director first sorts her file with $n \log(n)$ comparisons and then performs $n \log(n)$ more comparisons for each of the four payroll periods in one semester, how many comparisons are performed for 600 and for 1200 students? And for 1800 students?

The employment director finds she can't keep up with the explosion of work in her office as the number of employees doubles and then triples. She petitions the president's office for an assistant. The petition is granted, and the president even offers, if need be, to provide a second assistant. The next sections will present ways in which the director can use her assistants effectively.

EXERCISES FOR SECTION 5

1. Given an array of n distinct elements, we have said that there are $n!$ possible initial orderings of the n elements. For $n = 3$ and $n = 4$, give examples of $n!$ arrays, one corresponding to each of the possible orderings.
2. Suppose that in the array $A = (a_1, a_2, a_3, a_4)$ it is known that $a_3 = a_4$. Then how many different possible (unsorted) orderings are there?
3. Suppose that it is known that in the array $A = (a_1, a_2, \dots, a_n)$ two elements are the same but otherwise the elements are distinct. How many different (unsorted) orderings are there?
4. Suppose that we have an array (a_1, a_2, a_3) and ask the questions

“Is $a_3 \leq a_2$? and Is $a_2 \leq a_1$?”

Give an example of values for a_1 , a_2 , and a_3 for which the answers to these two questions do determine the order correctly. Then give an example of values for which the answers are not sufficient to determine the order correctly.

5. Suppose that we have an array (a_1, a_2, a_3, a_4) . Find four questions that sometimes do and sometimes do not determine the correct order.
6. If an algorithm makes k comparisons, explain why the corresponding binary tree has depth k .
7. Given an array containing n elements, is it ever possible to ask $n - 2$ or fewer questions of the form “Is $a_i \leq a_j$?” and from the answers to learn the correct order? Explain.
8. Suppose you have an n -element array and ask the $n - 1$ questions “Is $a_1 \leq a_2$?” “Is $a_2 \leq a_3$?” “Is $a_{n-1} \leq a_n$?” If it is possible to determine the order of the array from the answers to these questions, what can you say about the entries in the array?
9. Find a lower bound for $\log(n!)$ in the case that n is odd, similar to the one found in the text when n is even.
10. Stirling’s formula from Chapter 3 tells us that

$$\sqrt{n} \left(\frac{n}{e}\right)^n = O(n!).$$

Use this to obtain a lower bound on $\log(n!)$. Does this lead to a better (larger) lower bound than that derived in the text?

11. Draw a binary tree for sorting $A = (a_1, a_2, a_3, a_4)$ that begins by comparing a_1 with a_2 , then a_2 with a_3 , and a_3 with a_4 , and that has depth as small as possible. (As in Question 5.3 you may make different comparisons at different nodes on the same level.)
12. Suppose that a binary search tree for $A = (a_1, \dots, a_5)$ begins with the comparisons a_1 with a_2 , then a_1 with a_3 , then a_1 with a_4 and a_1 with a_5 . How many leaves are there at depth 4? Give an example of a leaf at depth 5. Give an example of a leaf at depth 6 or more.
13. Suppose that $tn \log(n)$ comparisons are made on records for n student employees during t time periods. If n is doubled to $2n$, does the number of comparisons double? Does it triple? Suppose that the number of students triples from n to $3n$? How many times the original number of comparisons is the new number of comparisons?
14. If $f(n) = n \log(n)$, find k such that for n sufficiently large

$$kn \log(n) \leq 2n \log(2n) \leq (k + 1)n \log(n).$$

15. Corollary 5.4 says that a file with five items in it cannot be sorted with fewer than $\lceil \log_2(5!) \rceil = 7$ comparisons. If one sorts a file with five records by first sorting four records and then inserting the fifth record, it takes a total of eight comparisons. First five comparisons are needed to correctly sort four records and then three more to insert the fifth item into the ordered list of four items. Decide whether the minimum number of comparisons that will necessarily sort a file with five items is seven or eight.
16. Here is a sorting game, played by two players on an array $A = (a_1, \dots, a_n)$. Player 1 picks two elements a_i and a_j and asks player 2 to compare these values and to say which is smaller. Player 2 then assigns values to a_i and a_j in any way and answers, for example, that $a_i \leq a_j$. Player 1 next picks another pair to compare, and player 2 again assigns values and reports the answer. (Once player 2 has picked and used a value for some a_i , the value cannot be changed, but values do not need to be selected until player 1 brings them up.) Player 1's goal is to determine the order as quickly as possible; player 2's goal is to keep the order obscure as long as possible. Play the sorting game for both $n = 4$ and $n = 5$. How many comparisons can player 2 force player 1 to make?
17. In the sorting game if player 1 asks for the results of k comparisons, then player 2 must give k different pieces of information, in this case either “<” or “>.” There are 2^k different patterns of answers that player 1 may receive from player 2. Use this to explain why to sort a set of n objects with k comparisons, it must be the case that $2^k \geq n!$.

6:6 RECURSION

The director of the student employment office hopes that with an assistant she can delegate more of the routine work. For example, at the beginning of the year she might sort half of the student records, give the other half to an assistant and then merge the two sorted files into one.

To train an assistant with an easy comparison task, the director begins with the job of finding the minimum entry of $A = (a_1, a_2, \dots, a_n)$, an array of real numbers. She asks the assistant to find the minimum entry of $A' = (a_1, a_2, \dots, a_{n-1})$ because she knows she can compare the assistant's minimum with a_n to find the overall minimum of A . Now the assistant catches on quickly and realizes that he can use another assistant to find the minimum of $A'' = (a_1, a_2, \dots, a_{n-2})$ and then compare that minimum with a_{n-1} to find the minimum of A' . If each of the assistants has an assistant (or a friend to help with the work), each assistant can pawn off the work of finding the minimum of a smaller array. Eventually, the array under consideration will have just one entry, which will be the minimum value of that array. This minimum value will be passed up and probably changed until the director receives the correct minimum of A' and then with one comparison finds the minimum of A .

This fanciful idea is an example of what is known as a recursive algorithm or recursive procedure. The word *recur* means to show up again, and that's exactly what happens in a recursive procedure: The procedure shows up again, or is used, within itself.

We now formalize a recursive procedure that carries out the idea described above. The procedure MIN will find the minimum entry in an array A . The input to MIN is A and n , the length of A , and the output from A is k , the value of the index of the minimum entry of A .

Procedure MIN(A, n, k)

```

STEP 1. If  $n = 1$ , then set  $k := 1$ 
        Else
        Begin
        STEP 2.  $n := n - 1$ 
        STEP 3. Procedure MIN( $A, n, k$ )
        STEP 4. If  $a_{n+1} < a_k$ , then  $k := n + 1$ 
        End {step 1}
STEP 5. Return.

```

In step 3 we call the procedure MIN, but on an array of smaller size. This is the essence of a recursive procedure. We repeatedly call MIN until $n = 1$. When $n = 1$, the array has one element and we actually find the minimum, successfully completing step 3. Every time step 3 is completed, we proceed to step 4 with the value of k just received and with the value of n equal to what it was when that instance of the procedure MIN was called.

Example 6.1. Table 6.4 is a trace of MIN with $A = (4.2, 2.1, 3.5, 0.9)$,

When $n = 1$, we set $k := 1$ and return to (C) to complete steps 4 and 5. Notice that when we then return to (B), n is reset to 2, its value at the time of this execution of step 3.

Question 6.1. Trace the procedure MIN on the array (4,3,2,1, 5).

There are two properties essential for a recursive procedure to be correct. These are dictated by the requirement that an algorithm must terminate after a finite number of steps. At each call of the procedure within the procedure, the value assigned to some variable, say P ($P = n =$ number of elements in the array in Example 6.1) must decrease. When the value assigned to this variable is sufficiently small, there must be a "termination condition" that instructs the procedure what to do in this final case. It is common to think of a recursive procedure as operating on different levels. If the procedure begins with the parameter P initially assigned the value q , one might think of beginning at the q th story of a building. With

Table 6.4

Step No.	n	k	a_k	a_{n+1}
1.	4			
2.	3			
3.	{Call MIN(<4.2, 2.1, 3.5), 3, k)}			(A)
1.	3			
2.	2			
3.	{Call MIN(<4.2, 2.1>, 2, k)}			(B)
1.	2			
2.	1			
3.	{Call MIN(<4.2>, 1, k)}			(c)
1.	1	1		
5.	{Return to (C)}			
	1	1	4.2	2.1
4.	1	2		
5.	{Return to (B)}			
	2	2	2.1	3.5
4.	2	2		
5.	{Return to (A)}			
	3	2	2.1	0.9
4.	3			
5.	{Return with $k^s = 4$ }.			

each call the value assigned to P is decreased, and one descends to a lower story until, say, P is assigned the value 1. On the first floor some real calculation or comparison is performed and the message is sent back up through the floors to the q th story, where the final answer is assembled.

A recursive program is also analogous to an induction proof. The “termination condition” corresponds to checking the base case. The call of the procedure within itself corresponds to using the inductive hypothesis.

Example 6.2. Here is an example of a recursive procedure that calculates the n th Fibonacci number (see Section 4.4).

Procedure FIB(n, F) { This procedure has n as input and F as output. }

```

STEP 1. If  $n \leq 1$ , then  $F := n$ 
        Else
        Begin
        STEP 2. Procedure FIB( $n - 1, F'$ )
        STEP 3. Procedure FIB( $n - 2, F''$ )
        STEP 4.  $F := F' + F''$ 
        End {step 1}
STEP 7. Return.
```

In step 1 we use the fact that $F_0 = 0$ and $F_1 = 1$. Notice that we can call FIB with input $n - 1$ or $n - 2$; the parameter n does not have to be decreased before the call, as was done in MIN. And the answers will be stored in F' and F'' as directed.

Another classic example of the use of recursive procedures is in calculating the greatest common divisor of two integers (see Algorithm EUCLID from Chapter 4). The next procedure is based on the fact that $\text{gcd}(b, c) = \text{gcd}(r, b)$, where $r = c - \lfloor c/b \rfloor b$. The procedure takes b and c as input and produces $g = \text{gcd}(b, c)$ as output.

Procedure GCD(b, c, g)

```

STEP 1. If  $b = 0$ , then  $g := c$ 
        Else
        Begin
        STEP 2.  $r := c - \lfloor c/b \rfloor * b$ 
        STEP 3. Procedure GCD( $r, b, g$ )
        End {step 1}
STEP 4. Return.

```

COMMENT. The values of b and c used in computing r in step 2 come from the input parameters of the procedure. They equal the original b and c only in the first execution of step 2.

Question 6.2. Trace GCD with $b = 13$ and $c = 21$. How many recursive calls does it make?

In the exercises you will see examples and problems on recursive procedures for the algorithms SUBSET, JSET, PERM, BtoD, among others. Some of these will be more efficient than before, others no more so.

We conclude with a recursive version of SELECTSORT based on an extension of MIN. The plan is to use basically the same ideas as in SELECTSORT, only to allow a director-sorter to delegate work to assistants. First we rewrite MIN so that upon input of an array A and two integers $\text{start} \leq \text{finish}$, it proceeds recursively to find the index k of the minimum entry in the subarray $\langle a_{\text{start}}, a_{\text{start}+1}, \dots, a_{\text{finish}} \rangle$.

Procedure MIN(A, start, finish, k)

```

STEP 1. If  $\text{start} = \text{finish}$ , then  $k := \text{start}$ 
        Else
        Begin
        STEP 2. Procedure MIN( $A, \text{start}, \text{finish} - 1, k$ )
        STEP 3. If  $a_{\text{finish}} < a_k$ , then  $k := \text{finish}$ 
        End {step 1}
STEP 4. Return.

```

Question 6.3. Trace MIN on $A = (-1, 0.333, 5.2, -10, 6.001, 17)$ for:
 (a) start = 2, finish = 3; (b) start = 3, finish = 6; and (c) start = 1, finish = 6.

Algorithm R-SELECTSORT

```

STEP 1. Input an array  $A$  and its length  $n$ 
STEP 2. For start := 1 to  $n - 1$  do
  Begin
  STEP 3. Procedure MIN( $A$ , start,  $n$ ,  $k$ )
  STEP 4. If  $k \neq$  start, then switch the values of  $a_{\text{start}}$  and  $a_k$ 
  End {step 2}
STEP 5. stop.
```

We don't claim that R-SELECTSORT is an improvement over SELECTSORT, but it is good training for the recursive sorting algorithm in the next section. In fact, R-SELECTSORT performs about twice as many comparisons as SELECTSORT as we shall see in the following discussion.

First we count the number of comparisons performed by MIN on an array of n elements, that is, when $n = \text{finish} - \text{start} + 1$; denote this number by $M(n)$. Then $M(1) = 1$. For $n > 1$, first one comparison is performed in step 1, then MIN is applied to an array with one fewer entry, and finally one additional comparison is made in step 3. Thus

$$M(n) = M(n - 1) + 2. \quad (\text{D})$$

In other words, each additional array entry requires two more comparisons. Thus

$$M(2) = M(1) + 2 = 3, \quad M(3) = M(2) + 2 = 5,$$

and apparently $M(n) = 2n - 1$. To be sure, we prove that this formula is correct by induction. Since $M(1) = 1$, the base case is correct. Then

$$\begin{aligned}
 M(n) &= M(n - 1) + 2 && \text{by (D)} \\
 &= 2(n - 1) - 1 + 2 && \text{by the inductive hypothesis} \\
 &= 2n - 1.
 \end{aligned}$$

Complexity results for recursive procedures are often similarly established using induction.

Now in R-SELECTSORT we call the procedure MIN(A , start, n , k) for start = 1, ..., $n - 1$. Thus the total number of comparisons performed is

$$(2n - 1) + (2n - 3) + \cdots + 3 = n^2 - 1 = O(n^2),$$

(see Exercise 14), giving the same big oh complexity as for SELECTSORT. Comparing the more precise count of comparisons (see Theorem 1.1) shows the recursive version to be less efficient.

EXERCISES FOR SECTION 6

1. Trace (the second version) of $\text{MIN}(A, 1, n, k)$ if
 - (a) $A = \langle -3, -2, -1 \rangle, n = 3.$
 - (b)** $A = \langle -10, 10, -3, 3 \rangle, n = 4.$
 - (c) $A = (1, 2, 3, 5, 7), n = 5.$
2. Trace GCD if
 - (a) $b = 3, c = 5.$
 - (b) $b = 1, c = 10.$
 - (c) $b = 0, c = 5.$
 - (d) $b = 3, c = 14.$
3. **(a)** Write a recursive procedure $\text{MINMAX}(A, n, \text{rein}, \text{max})$ to find the minimum and maximum entry of an array of n numbers.
(b) Determine the number of comparisons made in the worst case.
4. Using the fact that $n! = n(n-1)!$ write a recursive procedure $\text{FACT}(n, F)$ that upon input of a nonnegative integer n , calculates $F = n!$. Trace the procedure for $n = 4$.
5. What do the following recursive procedures compute?
 - (a) *Procedure NUM1(n, ans)*

STEP 1. If $n = 0$, then $\text{ans} := 0$
 Else
 Begin
 STEP 2. Procedure $\text{NUM1}(n + 1, \text{ans})$
 STEP 3. $\text{ans} := \text{ans} - (n + 1)$
 End
STEP 4. Return.

(b) *Procedure NUM2(n, ans)*

STEP 1. If $n = 0$, then $\text{ans} := 0$
 Else
 Begin
 STEP 2. Procedure $\text{NUM2}(n - 1, \text{ans})$
 STEP 3. $\text{ans} := \text{ans} + n$
 End
STEP 4. Return.

(c) *Procedure NUM3(n, ans)*

STEP 1. If $n = 0$, then $\text{ans} := 0$
 Else Procedure $\text{NUM3}(n - 2, \text{ans})$
STEP 2. Return.

6. Find an equation relating the two binomial coefficients $\binom{n}{k}$ and $\binom{n}{k-1}$. Use this to write a recursive procedure that calculates $\binom{n}{k}$. What is the termination condition?
7. Here is the classic relation between binomial coefficients:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Use this to write a recursive procedure to calculate $\binom{n}{k}$; you may need to use more than one termination condition.

8. Compare the algorithms in Exercises 6 and 7 by counting multiplications, divisions, additions, and the number of calls to the procedure. Which is more efficient?
9. Trace FIB with $n = 5$. Count the number of recursive calls. Comment on the efficiency of this method of calculating Fibonacci numbers as compared with methods learned in Chapter 4.
10. Suppose that $A(n)$ equals the number of additions performed in $\text{FIB}(n, F)$. Then $A(0) = A(1) = 0$ and $A(2) = 1$.
(a) Show that $A(n) = A(n-1) + A(n-2) + 1$.
(b) Compute $A(n)$ for $n = 3, 4, 5, 6, 7, 8$.
(c) Compare these values with the n th Fibonacci numbers for $n = 2, 3, \dots, 8$.
(d) Determine a formula for $A(n)$ and then prove that $\text{FIB}(n, F)$ performs this many additions.
(e) Is FIB a polynomial algorithm?
11. Suppose that we define $F_0^\# = 0$, $F_1^\# = 1$ and $F_2^\# = 1$, and for $n > 2$, $F_n^\# = F_{n-1}^\# + F_{n-2}^\# + F_{n-3}^\#$. Write a recursive procedure $\text{FIB}^\#(n, F^\#)$ that calculates $F_n^\#$ and stores it in $F^\#$.
12. Here is an attempt to improve the efficiency of the procedure FIB; to determine the n th Fibonacci number the numbers n , $s = 0$, and $t = 1$ should be input and F will be the output, containing the n th Fibonacci number.

Procedure FIB2(n, s, t, F)

STEP 1. If $n = 0$, then $F := s$
 Else Procedure **FIB2**($n - 1, t, s + t, F$)
STEP 2. Return.

Run this procedure with $n = 1, 2, 3, 4, 5$. Then explain why $\text{FIB2}(n, 0, 1, F)$ correctly returns F as the n th Fibonacci number.

13. Compare the number of recursive procedure calls made by FIB2 and by FIB.
14. Prove that $(2n - 1) + (2n - 3) + \cdots + 3 = n^2 - 1$.
15. Prove that the number of multiplications and divisions performed by the recursive version of GCD on b and c is at most $4\lfloor \log(b) \rfloor$. (*Hint*: Reread the complexity analysis of the algorithm EUCLID given in Chapter 4.)
16. Here is a recursive procedure to form a list L of all subsets of an n -set A .

Procedure R-SUBSET(A, n, L)

STEP 1. If $n = 0$, then $L := \{\emptyset\}$

Else

Begin

Procedure R-SUBSET($A, n - 1, L$)

STEP 2. For each set S in L , add $S \cup \{a(n)\}$ to L

End

STEP 3. Return.

(a) Trace this algorithm on $A = \{1, 2, \dots, n\}$ for $n = 2, 3$, and 4.

(b) Explain why the algorithm works correctly.

(c) If a step is considered to be the formation of a set, prove by induction that R-SUBSET(A, n, L) performs 2^n steps.

17. Here is a recursive version of the algorithm JSET, presented in Chapter 3. This procedure receives n and j and then stores all j -subsets of the n -set $\{1, 2, \dots, n\}$ in the list L .

Procedure R-JSET(n, j, L)

STEP 1. If $j = 0$, then $L := \{\emptyset\}$

Else if $j = n$, then $L := \{\{1, 2, \dots, n\}\}$

Else

Begin

STEP 2. Procedure R-JSET($n - 1, j, L_1$)

STEP 3. Procedure R-JSET($n - 1, j - 1, L_2$)

STEP 4. For each set S in L_2 ,

set $S := S \cup \{n\}$

STEP 5. $L := L_1 \cup L_2$

End

STEP 6. Return.

Run R-JSET on the following data: (a) $n = 2, j = 1$; (b) $n = 3, j = 1$; (c) $n = 3, j = 2$; (d) $n = 4, j = 2$; and (e) $n = 5, j = 3$.

18. What is the mathematical idea behind R-JSET that makes it work correctly? Count the number of assignment statements made in R-JSET. Is it a good algorithm?
19. Write a recursive version of TREESORT.

6:7 MERGESORT

In this section we present an efficient, recursive sorting algorithm, known as MERGESORT. This algorithm is particularly well suited to the situation when a set of records must be added to a large already sorted set of records. It has the disadvantage that to sort an array of n elements an additional array of size n is used to keep the programming simple and the element shuffling to a minimum.

Here is the idea of MERGESORT from the point of view of the employment director. Suppose that there is now enough work to employ two assistants. At the beginning of the semester the director receives a large file from the payroll office, containing one card for each student employee, listed in alphabetical order. She wants to sort these by social security number. To divide up the work, she splits the file, giving half to each assistant to sort. The director will then merge the two smaller sorted files into one large sorted file.

Algorithms that proceed by dividing the problem in half, working on one or both halves, and then constructing the final solution from the solutions to the smaller problems are known as divide-and-conquer algorithms. The algorithms BINARYSEARCH and BININSERT also follow this approach.

Back in the employment office, the assistants remember the principle of recursion. If they each have two assistants or friends, they will give half of their file to each for sorting and then merge the resulting sorted files. The halves or pieces to be sorted will get smaller until an array of one element is reached, say $\langle a \rangle$, and this array is sorted as it is. Notice that this process can be modeled by a binary tree with the root labeled with the director, the roots of the left and right subtrees labeled with the assistants, and so on.

Question 7.1. Let A be an array containing eight numbers. Using the ideas of the preceding paragraphs, draw the corresponding binary tree for this case. What is the depth of the tree? How many vertices does it contain? In total, how many assistants are employed in the task?

This approach will be good, provided that we can efficiently merge two sorted files into one sorted file. We shall see that such a merger can be performed in time linear in the total number of elements to be merged.

Here specifically is how to merge two sorted files. Assume that we have an array C such that $c_1, \dots, c_{\text{mid}}$ is in sorted order as is $c_{\text{mid}+1}, \dots, c_n$. (If we had two separate, sorted arrays A and B , we could place them in C with A listed before B .) The goal is to rearrange C so that it becomes a sorted array. We use an auxiliary array D into which we sort the elements of C ; in the end we transfer the sorted D back into C .

First we compare the first entries in the sorted subarrays, c_1 and $c_{\text{mid}+1}$, and place the smaller in d_1 . Next, depending on the outcome of the first comparison, we compare c_2 with $c_{\text{mid}+1}$ or c_1 with $c_{\text{mid}+2}$, placing the smaller in d_2 . We continue until either the first subarray or the second has been entirely placed in D .

Then we fill up D with the remaining elements of the other subarray and finally copy D into C .

Procedure MERGE (C , $start$, mid , $finish$) { C is an array with entries c_{start} , $c_{start+1}, \dots, c_{mid}$ in increasing order and entries $c_{mid+1}, \dots, c_{finish}$ also in increasing order.}

```

STEP 1.  Set  $i := start$  and  $j := mid + 1$  { $i$  and  $j$  index the entries of  $C$  being
        compared}
        Set  $k := start$  { $k$  indexes the entry of  $D$  being filled}
STEP 2.  While ( $i \leq mid$ ) and ( $j \leq finish$ ) do
STEP 3.  If  $c_i \leq c_j$ , then do
        Begin
STEP 4.   $d_k := c_i$ 
        STEPS.  $i := i + 1$ 
        STEP 6.  $k := k + 1$ 
        End
        Else
        Begin
STEP 7.   $d_k := c_j$ 
        STEP 8.  $j := j + 1$ 
        STEP 9.  $k := k + 1$ 
        End
        {Right now one of the subarrays is in  $D$ }
STEP 10. If  $i > mid$ , then do {Transfer remaining entries into  $D$ }
        For index :=  $j$  to  $finish$  do
        Begin
STEP 11.  $d_k := c_{index}$ 
        STEP 12.  $k := k + 1$ 
        End
        Else
        For index :=  $i$  to  $mid$  do
        Begin
STEP 13.  $d_k := c_{index}$ 
        STEP 14.  $k := k + 1$ 
        End
STEP 15. For index :=  $start$  to  $finish$  do {transfer  $D$  to  $C$ }
         $c_{index} := d_{index}$ 
STEP 16. Return.

```

Example 7.1. Table 6.5 is a trace of MERGE run on the array $C = (1, 2, 3, 4, -2, 0, 2, 4, 6)$ with $start = 1$, $mid = 4$, and $finish = 9$. We show the array D after the completion of each execution of step 3.

Table 6.5

<i>Step No.</i>	<i>i</i>	<i>j</i>	<i>k</i>	<i>D</i>
1	1	5	1	
3	1	6	2	(-2,...
	1	7	3	<-2,0,...
	2	7	4	(-2,0,1,...
	3	7	5	(-2,0,1,2,...
	3	8	6	(-2,0,1,2,2,...
4	4	8	7	(-2,0,1,2,2,3,...
	5	8	8	(-2,0,1,2,2,3,4,...
				<-2,0,1,2,2,3,4,4>6)
10				
15	$C = \langle -2, 0, 1, 2, 2, 3, 4, 4, 6 \rangle$.			

Notice that when equal entries occur, the entry of the first half is inserted in D first.

Question 7.2. Trace MERGE on $C = (0.1, 0.2, 0.3, 0, 0.09, 0.19, 0.29, 0.39, 0.49)$.

How **efficient** is MERGE? Three comparisons occur at every execution of steps 2 and 3, except for the final time when only the two comparisons in step 2 occur. These steps happen at most n times, where n is the length of the array. Then counting the additional comparison of step 10, at most $3n = O(n)$ comparisons are performed in total. MERGE is a linear algorithm.

With MERGE and the assurance of its efficiency, we plan MERGESORT. We begin with an unsorted array C of length n . We divide C at roughly the midpoint, setting mid equal to $\lfloor n/2 \rfloor$. We sort the first half of C recursively and the second half of C recursively and then use MERGE to combine them in sorted order. This will be accomplished by calling the recursive procedure below with $start = 1$ and $finish = n$.

Procedure MERGESORT ($C, start, finish$)

STEP 1. If $start = finish$, then Return.

Else

Begin

STEP 2. Set $mid := \lfloor (start + finish)/2 \rfloor$

STEP 3. Procedure MERGESORT ($C, start, mid$)

STEP 4. Procedure MERGESORT ($C, mid + 1, finish$)

STEP 5. Procedure MERGE ($C, start, mid, finish$)

End {step 1}

STEP 6. Return.

The main trick in tracing a procedure like this is to remember where to return upon the completion of a procedure and what the values assigned to the variables

are at the return. For example, if we call MERGESORT(C, i, j) in step 3, the procedure receives as input whatever subarray is currently stored in entries i through j of C , sorts it and returns it at the end of step 3 to the same subarray of C . All this bookkeeping is done for us in a programming language like Pascal.

Example 7.2. Table 6.6 is a trace of MERGESORT on $C = (0.3, 0.1, 0.2)$. The results of the procedure MERGE are just written under the call statement, since we have seen how this works before.

Procedure MERGESORT($\langle 0.3, 0.1, 0.2 \rangle, 1, 3$)

Table 6.6

<i>Step No.</i>	<i>c</i>	<i>start</i>	<i>mid</i>	<i>finish</i>	
1,2	(0.3,0.1,0.2)	1	2	3	
3	{Call MERGESORT($C, 1, 2$)}				(A)
1,2	(0.3,0.1)	1	1	2	
3	{Call MERGESORT($C, 1, 1$)}				(B)
1	(0.3)	1		1	
	Return to (B)	1	1	2	
4	{Call MERGESORT($C, 2, 2$)}				(c)
1	<0.1)	2		2	
	Return to (C)	1	1	2	
5	{Call MERGE($C, 1, 1, 2$)}				
	(0.1,0.3)				
	Return to (A)				
	(0.1,0.3, 0.2)	1	2	3	
4	{Call MERGESORT($C, 3, 3$)}				(D)
1	<0.2)	3		3	
	Return to (D)	1	2	3	
5	{Call MERGE($C, 1, 2, 3$)}				
	(0.1, 0.2, 0.3)				
6	Return.				

Question 7.3. Trace MERGESORT on

- (a) $C = \langle 1, 0 \rangle$ with $start = 1$ and $finish = 2$.
- (b) $C = (22, 24, 23)$ with $start = 1$ and $finish = 3$.
- (c) $C = \langle 1.1, 3.3, 2.2, 4.4 \rangle$ with $start = 1$ and $finish = 4$.

We now verify the efficiency of MERGESORT. The origins of this complexity bound are explored in Exercise 7.

Theorem 7.1. MERGESORT is a $O(n \log(n))$ algorithm.

Proof. We begin by proving that if $n = 2^k$, then the number of comparisons executed by MERGESORT is $3n \log(n) + 2n - 1$. The proof is by induction on k . If $k = 0$, then C contains one entry and with one comparison in step 1 the procedure is finished. Since $1 = 3 \cdot 1 \log(1) + 2 \cdot 1 - 1$, the base case is established.

We assume that the result holds for all exponents less than k and consider an array C with 2^k entries. Then initially mid equals 2^{k-1} , and in steps 3 and 4 MERGESORT is called on arrays of $n' = 2^{k-1}$ entries each. By the inductive hypothesis MERGESORT performs

$$\begin{aligned} 3n' \log(n') + 2n' - 1 &= 3 \cdot 2^{k-1} \log(2^{k-1}) + 2 \cdot 2^{k-1} - 1 \\ &= 3 \cdot 2^{k-1}(k-1) + 2 \cdot 2^{k-1} - 1 \\ &= 2^{k-1}(3k-1) - 1 \end{aligned}$$

comparisons on each smaller array. The total number of comparisons is 1 (from step 1) plus the number performed on the first half of C plus the number performed on the second half of C plus $3n$, the number of comparisons used by MERGE, or

$$\begin{aligned} 2(2^{k-1}(3k-1) - 1) + 3n + 1 &= 2^k(3k-1) + 3 \cdot 2^k - 1 \\ &= 2^k 3k + 2 \cdot 2^k - 1 \\ &= 3n \log(n) + 2n - 1. \end{aligned}$$

Now suppose that C is an array of n elements, where n is not necessarily a power of 2. Set $r = \lceil \log(n) \rceil$ and $m = 2^r$. We know

$$n \leq m = 2^r < 2^{\log(n)+1} = 2n.$$

Create C' , an array of m elements by appending $m - n$ new elements to the end of C . Suppose that all these elements are assigned a very large value, a number larger than all entries in C . When MERGESORT is applied to C' we know that the number of comparisons is at most

$$\begin{aligned} 3m \log(m) + 2m - 1 &< 3 \cdot (2n) \log(2n) + 2 \cdot (2n) - 1 = 6n(\log(n) + 1) + 4n - 1 \\ &= 6n \log(n) + 10n - 1 \\ &< 16n \log(n) = O(n \log(n)). \end{aligned}$$

Now C' and C have been sorted with $O(n \log(n))$ comparisons; had we applied MERGESORT to C alone, perhaps fewer comparisons would have been performed. \square

6 SEARCHING AND SORTING

An alternative, tighter upper bound on the number of comparisons in MERGESORT is outlined in Supplementary Exercises 27 and 28.

Question 7.4. Look at Question 7.3(a) and (c) and verify that exactly $3n \log(n) + 2n - 1$ comparisons were performed.

Question 7.5. Verify that the number of comparisons MERGESORT performs on an array of size 3 is 20. Is this number less than $3n \log(n) + 2n - 1$ with $n = 3$? Show that this number is less than $6n \log(n) + 10n - 1$ when $n = 3$.

EXERCISES FOR SECTION 7

- Trace MERGE on the following data. In each case count the number of comparisons made and compare with $3n$, where n is the length of the array.
 - $C = (1, 1, 3, 5)$, start= 1, mid= 1 and finish= 4.
 - $C = (0.1, 0.2, 0.3, 0.2, 0.4, 0.6)$, start= 1, mid= 3 and finish= 7.
 - $C = (1, 2, 3, 4, 1, 2, 3, 4)$, start= 1, mid= 4 and finish= 8.
 - $C = (5, 1, 2, 3, 4)$, start= 1, mid= 1 and finish= 5.
 - $C = (1, 2, 3, 4, 5)$, start= 1, mid= 4 and finish= 5.
 - $C = \langle 1, 2, 3, 4, 0 \rangle$, start = 1, mid = 4 and finish = 5.
- What happens if you run MERGE with the subarray $c_1, \dots, c_{\text{mid}}$ not sorted?
- Trace MERGESORT on each of the following arrays:
 - $A = (2, 4, 6, 8, 10)$.
 - $A = (10, 8, 6, 4, 2)$.
 - $A = (2, 6, 4, 10, 8)$.
 - $A = \langle 1, 3, 1, 5, 4, 5 \rangle$.
 - $A = (2, 2, 2, 2, 2)$.
- Draw a binary tree that corresponds to the divisions into subarrays in Example 7.2 and Question 7.3. In general, for what arrays of length n is the corresponding tree a full binary tree?
- Count the number of comparisons made in each case of Exercise 3. Compare these numbers with $3n \log(n) + 2n - 1$ and with $6n \log(n) + 10n - 1$ for appropriate values of n .
- From the numerical evidence of Questions 7.4 and 7.5 and Exercise 3, conjecture whether the following is true or false: MERGESORT performs at most $3n \lceil \log(n) \rceil + 2n - 1$ comparisons to sort an array of n elements. (See also Supplementary Exercises 27 and 28.)
- Let $M(n)$ denote the maximum number of comparisons made in MERGESORT, applied to an array of n elements, and suppose that $n = 2^k$. Then $M(1) = 1$ and for $n > 1$ MERGESORT proceeds by calling itself on two arrays

of size $n/2 = 2^{k-1}$ and then MERGE-ing the two sorted arrays with $3n$ additional comparisons. Thus

$$M(n) = 2M(n/2) + 3n + 1.$$

Use this relation to determine $M(n)$ for $n = 2, 4, 8$, and 16.

Then find an expression for ' $M(n)$ in terms of $M(n/4)$ and in terms of $M(n/8)$. Explain why this leads to the formula

$$\begin{aligned} M(n) &= 2^k M(1) + k3n + n - 1 \\ &= 3n \log(n) + 2n - 1 \quad (\text{still assuming that } n = 2^k). \end{aligned}$$

8. Suppose that A is an array containing 2^n numbers. If the array is divided in half for each of two assistants to sort and if they each divide their half in half for two additional assistants to sort, and so on, until finally the assistants receive arrays of length one, then how many assistants in all are used? How many levels of assistants are used?
9. Answer the same question as in Exercise 8 when the array contains m numbers, where $2^n \leq m < 2^{n+1}$ for some integer n .
10. Write a procedure that inputs an array $A = \langle a_1, a_2, \dots, a_n \rangle$ (not necessarily sorted) and rearranges A so that if $\text{mid} = \lfloor (1+n)/2 \rfloor$ and $S = a_{\text{mid}}$, then all entries preceding S are less than or equal to S and all entries following S are greater than or equal to it. (Note that S may need to be moved to a different position.)
11. Write a sorting algorithm that splits the input array A using the preceding exercise and then recursively sorts the parts preceding and following S .
12. Write an algorithm 4-MERGE that takes four sorted arrays and merges them into one sorted array. Compare the complexity of your algorithm with that of using MERGE three times to combine these four arrays into one.

6:8 SORTING IT ALL OUT

The art of searching and sorting is an extremely important and highly developed one in computer science and applications. These are processes used in almost all record-keeping tasks. Not only do telephone companies, banks, the IRS, and so forth, perform these tasks repeatedly, but now even writers find these tasks indispensable in their word processing programs. For example, searching was done repeatedly in the preparation of this text. Every time a theorem, a question, an example, or an exercise was renumbered, a search was run to find all occurrences of the changed number. A spelling checker program also searches for spelling

errors and is equally useful because it picks up most typing mistakes. Sorting is also important, for example, in alphabetizing the index of a book.

One important theme of this chapter is the difference between $O(n^2)$ and $O(n \log(n))$ algorithms. Both kinds are good algorithms, but the latter are noticeably more efficient. Except for cases with small data sets (small like most that we've considered in examples and exercises), the faster algorithms make a significant difference in general real-life applications. Of course, there are exceptions to every rule, and two such exceptions are BUBBLESORT (Exercises 1.11 to 1.13) and INSERTIONSORT (Supplementary Exercise 7). In the worst case these are $O(n^2)$ algorithms, but when given a nearly sorted array, they can run in linear time. For example, when an array is nearly sorted except that some adjacent pairs of elements are transposed, both BUBBLESORT and INSERTIONSORT are able to benefit from the nearly sorted arrangement. In contrast an algorithm like BINARYSORT will perform the same number of comparisons on a nearly sorted array as on a randomly ordered array.

Another theme of this chapter is the difference between algorithms with the same big oh complexity. When sorting files with large individual records, algorithms should be used that minimize record transfers; for example, SELECTSORT would be preferred to BUBBLESORT if a $O(n^2)$ algorithm were being used. MERGESORT should be avoided if the file length is so long or the records so large that there is not room for a duplicate array. However, MERGESORT is a good choice when two smaller sorted files are to be sorted into one. No algorithm using comparisons can be faster than $O(n \log(n))$, and the number of different $n \log(n)$ algorithms confirms that these must have been designed for varying needs. TREESORT uses the most sophisticated data structure among the algorithms we've seen. This algorithm and algorithms based on storing data in tree structures have wide applicability in these and other combinatorial settings.

We have alluded to the existence of a linear-time sorting algorithm, known as BUCKETSORT or "distribution counting." If we want to sort an array of n elements whose entries are integers from 0 to M for some small number M , like $M = O(n)$, then we can make one pass through the array and can store the record with key a_i in the a_i th entry of a new array. (More picturesquely, we think of tossing the record in the a_i th "bucket.") Then in one additional sweep through the new array, we can pick up the elements in order. We have performed $n + M$ assignments and no comparisons. This algorithm has limited applicability; for example, using this algorithm the employment director would store the record cards of, say, 600 students in a new array of length 999,999,999, since there are this many possible social security numbers. This approach would necessitate an inappropriately large array. More sophisticated versions of such an algorithm are known as hashing.

Finally, the concept of recursive procedures is an important one. This is the computer scientists' analogue of induction. In Chapter 7 we shall study solutions of recurrence relations and counting problems that arise from recursive procedures.

This chapter has been only an introduction to a deep and well-understood theory, which merits further study.¹

SUPPLEMENTARY EXERCISES FOR CHAPTER 6

1. Devise an algorithm TRISECTSEARCH that upon input of an array A of n numbers in increasing order and a number S , searches by thirds of A for S . Specifically, first the algorithm should see whether S equals the $\lfloor (n+1)/3 \rfloor$ rd entry in A . If not and S is smaller, then it begins again with the first third of A . If S is larger than this entry, it compares S with the $\lfloor 2(n+1)/3 \rfloor$ rd entry. If S is smaller, it proceeds with the middle third of A ; if S is larger, it proceeds with the last third of A . Determine the worst-case complexity of your algorithm.
2. Write an algorithm that upon input of an ordered array A of n numbers and a number S , searches for S and if found, deletes it. Determine the worst-case complexity of your algorithm.
3. Write an algorithm that upon input of an ordered array A of n numbers and a number S , searches for S and if it is not found, inserts it in the correct order. Determine the worst-case complexity of your algorithm.
4. Rewrite a version of BININSERT, called BININSERT2, that tests whether $a_{r+1} = a_{\text{mid}}$ after step 3, and if so, immediately inserts a_{r+1} at the (mid)th entry of the array. Are there arrays on which BININSERT2 will run faster than BININSERT? Are there arrays on which BININSERT2 will run slower than on BININSERT? Determine the worst-case complexity of BININSERT2.
5. Use BININSERT2 to form a new version of BINARYSORT, called BINARYSORT2. Run both BINARYSORT and BINARYSORT2 on $\langle 1,5,1, 1,1 \rangle$ and compare the efficiency of these algorithms on this array.
6. In this exercise you are asked to compare the number of assignment statements in SELECTSORT and in BINARYSORT. The significant assignment statements are those involving array elements, not just index counters in loops.
 - (a) Rewrite SELECTSORT so that step 5 is expanded and actually carries out the details of switching a_i and TN . Call this X-SELECTSORT.
 - (b) Count the number of assignments of elements a_i and TN in the worst case in X-SELECTSORT.
 - (c) In the procedure BININSERT with $r = 1,2,3$, and 4 find examples in which $r + 2$ assignments of elements a_i and temp are made.
 - (d) Explain why the maximum number of assignments of elements a_i and temp in BININSERT is $r + 2$.

¹ A good next source is a large book on the subject: D. E. Knuth, *Sorting and Searching*, Volume 3 of *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1973.

- (e) Use the result of part (d) to determine the maximum number of assignment statements performed in BINARYSORT.
- (f) Which of X-SELECTSORT and BINARYSORT performs more assignments?
7. INSERTIONSORT is another sorting algorithm; it is based on the idea of how one often sorts a hand of playing cards: with the left end of the hand sorted, the remaining cards are inserted *in* order, one at a time.
- (a) Write a procedure INSERT($r, a_1, \dots, a_r, a_{r+1}$) that has a sorted array of length r , $\langle a_1, a_2, \dots, a_r \rangle$, and an element a_{r+1} as input and that outputs the array (a_1, \dots, a_{r+1}) in sorted order. The procedure should search through the input array sequentially until the position for inserting a_{r+1} is found; then a_{r+1} should be inserted there.
- (b) Here is the algorithm INSERTIONSORT:

Algorithm INSERTIONSORT

- STEP 1. Input n and an array $\langle a_1, a_2, \dots, a_n \rangle$
 STEP 2. For $m = 2$ to n do {insert m th entry}
 STEP 3. Procedure INSERT $((m-1), a_1, \dots, a_m)$
 STEP 4. Stop.

Trace this on (1,3,2,5,4, 6).

- (c) Compare this sorting algorithm with SELECTSORT and BINARYSORT. Describe arrays on which INSERTIONSORT works more efficiently than the others and arrays on which it is less efficient.
- (d) Determine the complexity of INSERTIONSORT.
8. A sorting algorithm is said to be stable if whenever $a_i = a_j$ for some indices $i < j$, then in the sorted array a_i precedes a_j . Is either SELECTSORT or BINARYSORT stable? Explain. If not, can they be rewritten (easily) so that they are stable?
9. Is INSERTIONSORT a stable sorting algorithm?
10. Suppose that A is a sorted array of n elements. How does the speed of INSERTIONSORT on A compare with the speed of SELECTSORT and BINARYSORT?
11. Suppose that you have an (unsorted) array with n items and another item D . What is the minimum number of comparisons necessary to determine whether D is contained in the array or not?
12. Looking up a telephone number in a directory is an example of a typical search through a large ordered list. If the name you are looking for is, say, Smith, you wouldn't turn to the exact middle of the directory despite the high quality of BINARYSEARCH. The reason is that you have some knowledge concerning how the names in the directory are distributed. If you are looking

for the name Smith, you will look more toward the back of the book because you expect that more names come before Smith than after it. You might use a strategy like the following: Since S is the 19th letter of the alphabet, you might look at the page numbered m , where $m = \sim 19n/26J$ and n is the total number of pages in the directory. Develop an algorithm, called weighted binary search, that exploits this idea. When should you use weighted binary search and when should you definitely avoid it? (This kind of approach is also known as interpolated search.)

13. Here is a recursive version of the algorithm **DtoB** from Chapter 1 that upon input of a nonnegative integer m determines its binary expansion s .

Procedure R-DtoB(m, s)

STEP 1. If $m \leq 1$, then set $s := m$

 Else

 Begin

 STEP 2. Procedure **R-DtoB**($\lfloor m/2 \rfloor, s$)

 STEP 3. If m is even, then sets equal to s with a 0 added at the end,
 Else set s equal to s with a 1 added at the end

STEP 4. Return.

(a) Trace this algorithm for $m = 1, 3, 6, 8$.

(b) Show that this algorithm is correct.

(c) Prove by induction that the number of divisions in **R-DtoB** is at most $\log(m)$.

14. Reread the algorithm **EXPONENT** in Chapter 2. Then use the fact that $x^a = x \cdot x^{a-1}$ to write a recursive version of **EXPONENT**. Compare the number of multiplications in **EXPONENT** and the recursive version.
15. Write a recursive version of **FASTEXP**, called **R-FASTEXP**(X, n, ans) that upon input of x and n will calculate X^n and store it in ans . Is this version faster or slower than **FASTEXP**?
16. Look back in Chapter 3 at the algorithm **PERM**. Write a recursive version of this algorithm.
17. Does the following correctly compute the greatest common divisor of b and c ? Explain.

Procedure GCD3(b, c, g)

STEP 1. If $b = c$, then $g := b$

 Else if $b \leq c - b$, then Procedure **GCD3**($b, c - b, g$)

 Else Procedure **GCD3**($c - b, b, g$)

STEP 2. Return.

18. Here is an idea for a recursive version of BINARYSEARCH: Given an array (a_1, \dots, a_n) of numbers and a number S , determine whether S is less than the middle entry of the array and, if so, search the first half. If not, search the second half. Write a recursive version of BINARYSORT.
19. Suppose that we have a sorted array A of length n and an unsorted array B of length m that we wish to merge into A to form a final sorted array A of length $n + m$. Here are some different approaches:
- (a) Add B to the end of the array A and then use BINARYSORT on this array.
 - (b) Add B to the end of the array A and then use MERGESORT on this array.
 - (c) Add B to the end of the array A and then use INSERTIONSORT (see Exercise 7), replacing step 2 with "For $m = n + 1$ to $n + m$ do."
 - (d) Use MERGESORT on B and then use MERGE on A and B .
- Comment on the pros and cons of these approaches. In particular, decide which one you would pick for best efficiency.
20. Compare the efficiency (i.e., number of comparisons performed) of SELECT-SORT, BINARYSORT, and MERGESORT on the following types of arrays:
- (a) A sorted array.
 - (b) An array listed in reverse order.
 - (c) An array that is nearly sorted except for the interchange of some adjacent pairs of numbers (like $(1, 3, 2, 5, 4, 6)$).
 - (d) An array with many repeated numbers.
 - (e) An array with its first half sorted and its second half sorted.
21. Is MERGESORT a stable sorting algorithm? (See Exercise 8.)
22. Develop the following idea into an algorithm to sort $A = (a_1, a_2, \dots, a_n)$.
- (1) Find the least integer i such that $\langle a_1, \dots, a_i \rangle$ is sorted, but $\langle a_1, \dots, a_i, a_{i+1} \rangle$ is not.
 - (2) Find the next least integer j such that (a_{j+1}, \dots, a_j) is sorted, but $\langle a_{i+1}, \dots, a_j, a_{j+1} \rangle$ is not.
 - (3) Merge (a_1, \dots, a_i) and (a_{i+1}, \dots, a_j) .
 - (4) Set $i := j$ and if $j < n$, go to line 2.
- Implement this as an algorithm and run it on the following data:
- (a) $A = (1, 3, 2, 5, 4, 6)$.
 - (b) $A = (1, 2, 3, 5, 4, 6)$.
 - (c) $A = (2, 4, 6, 3, 5, 7)$.
 - (d) $A = (1, 2, 3, 4, 5, 6)$.
 - (e) $A = (6, 5, 4, 3, 2, 1)$.
- Determine the worst-case complexity of your algorithm.
23. Here is an alleged sorting algorithm that is supposed to take an array of length n with $\text{start} = 1$ and $\text{finish} = n$ and to rearrange A in increasing order:

Procedure *MYSTERY*(*A*, *start*, *finish*)

STEP 1. If *start* < *finish*, then
 Begin
 STEP 2. $\text{test} := a_{\text{start}}$
 STEP 3. $i := \text{start} + 1$
 STEP 4. $j := \text{finish}$
 STEP 5. Repeat
 Begin
 STEP 6. While $\text{test} < a_j$, $\text{set } j := j - 1$
 STEP 7. While $\text{test} > a_i$ and $i < \text{finish}$, $\text{set } i := i + 1$
 STEP 8. Switch a_i and a_j
 Until $j \leq i$ {end of step 5}
 STEP 9. Switch a_i and a_j {undoing the last switch}
 STEP 10. Switch a_{start} and a_j
 STEP 11. Procedure *MYSTERY*(*A*, *start*, $j - 1$)
 STEP 12. Procedure *MYSTERY*(*A*, $j + 1$, *finish*)
 End {step 1}
 STEP 13. Return.

Run this algorithm on a variety of arrays and then answer the following equations:

- (a) *MYSTERY* finds an index *j*, places some entry in it, and then recursively goes to work on the array in front of *j* and behind *j*. What value of *j* does it determine and what entry is placed in a_j ?
- (b) Describe in words how *MYSTERY* works.
- (c) Determine the worst-case complexity of *MYSTERY*.
24. Why does *BINARYSEARCH* require more comparisons than *BININSERT* in the worst case?
25. Rewrite *BINARYSEARCH* so that the maximum number of comparisons it performs in searching an array of *n* items is $2 \log(n) + c$, where *c* is a constant.
26. In the complexity analysis of *BINARYSORT* we proved that the maximum number of comparisons performed on an array of *n* elements is $4(n - 1) + 2 \log((n - 1)!)$. First prove that for $i = 1, \dots, n - 1$,

$$i(n + 1 - i) \leq \frac{(n + 1)^2}{4}.$$

Then use this to derive an upper bound on $\log((n - 1)!)$ that is $O(n \log(n))$.

27. Let $T(n)$ denote the maximum number of comparisons performed by *MERGE-SORT* on an array of *n* entries. Then explain why

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 3n + 1 \quad \text{for } n > 1$$

6 SEARCHING AND SORTING

and

$$T(1) = 1.$$

Calculate $T(i)$ for $i \leq 8$ and compare these results with those of Questions 7.4 and 7.5 and Exercise 7.5.

28. Use the results of the previous exercise to prove that

$$T(n) \leq 3n \lceil \log(n) \rceil + 2n - 1.$$

29. Suppose that a file of n records is to be sorted and the keys of these records are known to be precisely the numbers $1, 2, \dots, n$. Here is an algorithm to accomplish a sort on the keys a_i :

Algorithm BUCKETSORT

STEP 1. Input $A = \langle a_1, a_2, \dots, a_n \rangle$ containing distinct entries from $1, 2, \dots, n$

STEP 2. For $i := 1$ to n do

 STEP 3. $B(a_i) := a_i$

STEP 4. For $i := 1$ to n do

 STEP 5. $a_i := B(i)$

STEP 6. Output $\langle a_1, a_2, \dots, a_n \rangle$.

Run a trace on this algorithm with input $A = (2, 1, 5, 4, 3, 6, 7)$.

30. Count the number of assignment statements made in BUCKETSORT when run on an array of size n ; in this algorithm these are the most time-consuming statements.
31. Write an algorithm BUCKETSORT2 that has as input an array A of n distinct numbers whose entries lie between 0 and some constant M . The algorithm should first do a “bucketsort” of A into an array B of length M and then transfer the sorted elements back into A . Count the number of assignment statements made in BUCKETSORT2.
32. Suppose that a comparison takes twice as long as an assignment statement. Compare the time needed to run BUCKETSORT2 when $M = 2n$, kn for some constant k , $n \log(n)$, and n^2 with the time needed for BINARYSORT.
33. The Pancake Problem asks the following Given a stack of pancakes of varying diameters, rearrange them into a stack with decreasing diameter (as you move up the stack) using only “spatula flips.” With a spatula flip you insert the spatula and invert the (sub)stack of pancakes above the spatula. Design an algorithm that correctly solves the pancake problem for a stack of n pancakes with at most $2n$ flips. Count exactly how many flips your algorithm uses in the worst case.