CHAPTER 6

Sorting

Sorting and searching are among the most typical functions of programming systems. In the first section of this chapter we discuss some of the considerations involved in sorting. In the remainder of the chapter we discuss some of the more common sorting techniques and the advantages or disadvantages of one technique over another. In the next chapter we discuss searching and some applications.

6.1 GENERAL BACKGROUND

The concept of an ordered set of elements has a major impact on our daily lives. Consider, for example, the process of finding a telephone number in a telephone directory. This process, called a ***search***, is greatly simplified by the fact that the names in the directory are listed in alphabetical order. Imagine how much trouble you would have in locating a telephone number if the names were listed in the order in which the customers obtained their phone service from the telephone company. In such a case, the names might just as well be entered in random order. Sorting the entries in alphabetical rather than chronological order simplifies the process of searching. Or, imagine the case of someone searching for a book in a library. Since the books are shelved in a specific order (Library of Congress, Dewey System, etc.), each book is assigned a specific position relative to the others and can be retrieved in a reasonable amount of time (if it is there). Or, consider a set of numbers sorted sequentially in a computer's memory. As we shall see in the next chapter, it is usually easier to find an element of a set if the numbers are maintained in sorted order. In general, a set of items is kept sorted in order to either produce a report (i.e., to simplify manual retrieval of information, as in a telephone book or a library shelf) or to make machine access to data more efficient.

We now present some basic terminology. A ***file*** of size n is a sequence of n items $r[0], r[1], \ldots, r[n-1]$. Each item in the file is called a ***record***. A key, $k[i]$, is associated with each record $r[i]$. The key is usually (but not always) a subfield of the entire record. The file is said to be ***sorted on the key*** if $i < j$ implies that $k[i]$ precedes $k[j]$ in some ordering on the keys. In the example of the telephone book, the file consists of all the entries in the book. Each entry is a record. The key upon which the file is sorted is the

name field of the record. Each record also contains fields for an address and a telephone number. A ***sort*** is a process that rearranges the records in a file into a sequence that is sorted on some key.

A sort can be classified as ***internal*** if the records it is sorting are in main memory and as ***external*** if some of the records it is sorting are in auxiliary storage. We restrict our attention to internal sorts.

It is possible for two records in a file to have the same key. A sorting technique is called ***stable*** if for all records i and j such that $k[i]$ equals $k[j]$, if $r[i]$ precedes $r[j]$ in the original file, then $r[i]$ precedes $r[j]$ in the sorted file.

A sort takes place either on the records themselves or on an auxiliary table of pointers. For example, a file of five records is shown in Figure 6.1.1a. If the file is sorted in increasing order on the numeric key shown, then the resulting file is as shown in Figure 6.1.1b. In this case the actual records themselves have been sorted.

Suppose, however, that the amount of data stored in each of the records in the file in Figure 6.1.1a is so large that the overhead involved in moving the actual data is prohibitive. In this case, an auxiliary table of pointers may be used, so that the pointers are moved instead of the actual data, as shown in Figure 6.1.2. (This is called ***sorting by***

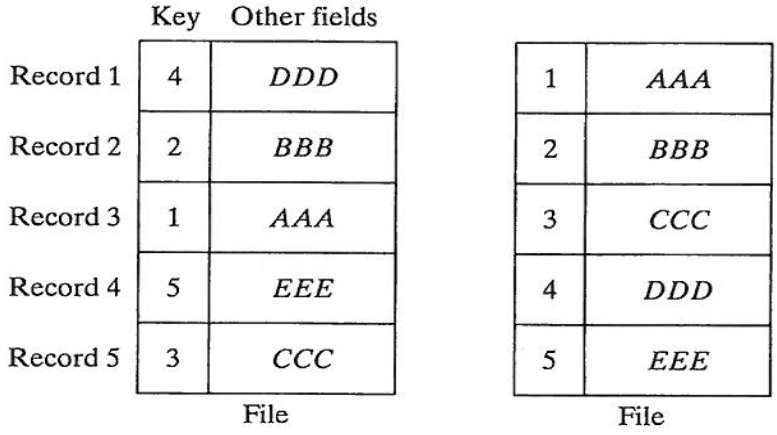

FIGURE 6.1.1 Sorting actual records.

(a) Original file.

(b) Sorted file.

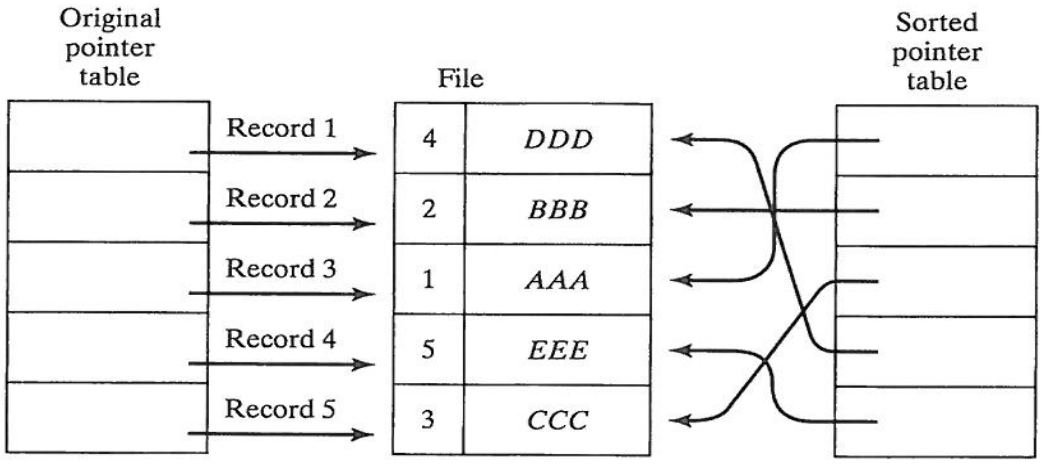

FIGURE 6.1.2 Sorting by using an auxiliary table of pointers.

address.) The table in the center is the file, and the table at the left is the initial table of pointers. The entry in position *j* in the table of pointers points to record *j*. The entries in the pointer table are adjusted during the sorting process so that the final table is as shown at the right. Originally, the first pointer was to the first entry in the file; upon completion, it is to the fourth entry in the table. Note that none of the original file entries are moved. In most of the programs in this chapter, we illustrate techniques for sorting actual records. The extension of these techniques to sorting by address is straightforward and will be left as an exercise for the reader. (Actually, for the sake of simplicity, we sort only the keys in the examples presented in this chapter; we leave it to the reader to modify the programs to sort full records.)

Because of the relationship between sorting and searching, the first question to ask in any application is whether or not a file should be sorted. Sometimes, there is less work involved in searching a set of elements for a file than in first sorting the entire set and then extracting the desired element. On the other hand, if frequent use of the file is required for the purpose of retrieving specific elements, then it might be more efficient to sort the file. This is because the overhead of successive searches may far exceed the overhead involved in sorting the file once and subsequently retrieving elements from the sorted file. Thus it cannot be said that it is more efficient either to sort or not to sort. The programmer must make a decision based on individual circumstances. Once a decision to sort has been made, other decisions must be made, including what is to be sorted and what methods are to be used. There is no one sorting method that is universally superior to all others. The programmer must carefully examine the problem and the desired results before deciding these very important questions.

Efficiency Considerations

As we shall see in this chapter, there are a great number of methods which can be used to sort a file. The programmer must be aware of several interrelated and often conflicting efficiency considerations to make an intelligent choice as to which sorting method is most appropriate to a particular problem. Three of the most important of these considerations are: the length of time the programmer must spend in coding a particular sorting program, the amount of machine time necessary for running the program, and the amount of space necessary for the program.

If a file is small, sophisticated sorting techniques designed to minimize space and time requirements are usually worse or only marginally better in achieving efficiencies than simpler, generally less efficient methods. Similarly, if a particular sorting program is to be run only once, and there is sufficient machine time and space in which to run it, it would be ludicrous for a programmer to spend days investigating the best methods of obtaining the last ounce of efficiency. In such cases, the amount of time which must be spent by the programmer is properly the overriding consideration in determining which sorting method to use. However, a strong word of caution must be inserted. Programming time is never a valid excuse for using an incorrect program. A sort that is run only once may be able to afford the luxury of an inefficient technique, but it cannot afford an incorrect one. The presumably sorted data may be used in an application in which the assumption of ordered data is crucial.

However, a programmer must be able to recognize the fact that a particular sort is inefficient and be able to justify its use in a particular situation. Too often,

programmers take the easy way out and code an inefficient sort which is then incorporated into a larger system in which the sort is a key component. The designers and planners of the system are then surprised at the inadequacy of their creation. In order to maximize their own efficiency, programmers must be familiar with a wide range of sorting techniques and be cognizant of the advantages and disadvantages of each, so that when the need for a sort arises they can supply the one that is most appropriate for the situation.

This brings us to the other two efficiency considerations: time and space. As in most other computer applications, the programmer must often optimize one of these at the expense of the other. In considering the time necessary to sort a file of size n, we do not concern ourselves with actual time units, because these will vary from one machine to another, from one program to another, and from one set of data to another. Rather, we are interested in the corresponding change in the amount of time required to sort a file induced by a change in the file size n. Let us see if we can make this concept more precise. We say that y is ***proportional*** to x if the relation between y and x is such that multiplying x by a constant multiplies y by the same constant. Thus if y is proportional to x, doubling x will double y, and multiplying x by 10 will multiply y by 10. Similarly, if y is proportional to x^2, then doubling x will multiply y by 4, and multiplying x by 10 will multiply y by 100.

Often we do not measure the time efficiency of a sort by the number of time units required but by the number of critical operations performed. Examples of such critical operations are key comparisons (i.e., comparisons of the keys of two records in the file to determine which is greater), movements of records or pointers to records, and interchanges of two records. The critical operations chosen are those that take the most time. For example, a key comparison may be a complex operation, especially if the keys are long or the ordering among them is nontrivial. Thus a key comparison requires much more time than, say, a simple increment of an index variable in a *for* loop. Also, the number of simple operations required is usually proportional to the number of key comparisons. For this reason, the number of key comparisons is a useful measure of a sort's time efficiency.

There are two ways to determine the time requirements of a sort, neither of which yields results that are applicable to all cases. One method is to go through a sometimes intricate and involved mathematical analysis of various cases (e.g., best case, worst case, average case). The result of this analysis is often a formula giving the average time (or number of operations) required for a particular sort as a method of the file size n. (Actually, the time requirements of a sort depend on factors other than file size, but we are concerned here only with dependence on file size.) Suppose that such a mathematical analysis on a particular sorting program results in the conclusion that the program takes $0.01n^2 + 10n$ time units to execute. The first and fourth columns in Figure 6.1.3 show the time needed by the sort for various values of n. You will notice that for small values of n, the quantity $10n$ (third column in Figure 6.1.3) overwhelms the quantity $0.01n^2$ (second column). This is because the difference between n^2 and n is small for small values of n and is more than compensated for by the difference between 10 and 0.01. Thus, for small values of n, an increase in n by a factor of 2 (e.g., from 50 to 100) increases the time needed for sorting by approximately that

n	$a = 0.01\,n^2$	$b = 10n$	$a + b$	$\frac{(a+b)}{n^2}$
10	1	100	101	1.01
50	25	500	525	0.21
100	100	1,000	1,100	0.11
500	2,500	5,000	7,500	0.03
1,000	10,000	10,000	20,000	0.02
5,000	250,000	50,000	300,000	0.01
10,000	1,000,000	100,000	1,100,000	0.01
50,000	25,000,000	500,000	25,500,000	0.01
100,000	100,000,000	1,000,000	101,000,000	0.01
500,000	2,500,000,000	5,000,000	2,505,000,000	0.01

FIGURE 6.1.3

same factor of 2 (from 525 to 1100). Similarly, an increase in n by a factor of 5 (e.g., from 10 to 50) increases the sorting time by approximately 5 (from 101 to 525).

However, as n becomes larger, the difference between n^2 and n increases so quickly that it eventually more than compensates for the difference between 10 and 0.01. Thus, when n equals 1000, the two terms contribute equally to the amount of time needed by the program. As n becomes even larger, the term $0.01n^2$ overwhelms the term $10n$, and the contribution of the term $10n$ becomes almost insignificant. Thus, for large values of n, an increase in n by a factor of 2 (e.g., from 50,000 to 100,000) results in an increase in sorting time of approximately 4 (from 25.5 million to 101 million), and an increase in n by a factor of 5 (e.g., from 10,000 to 50,000) increases the sorting time by approximately a factor of 25 (from 1.1 million to 25.5 million). Indeed, as n becomes larger and larger, the sorting time becomes more closely proportional to n^2, as is clearly illustrated by the last column in Figure 6.1.3. Thus, for large n the time required by the sort is almost proportional to n^2. Of course, for small values of n, the sort may exhibit drastically different behavior (as in Figure 6.1.3), a situation which must be taken into account in analyzing its efficiency.

O Notation

To capture the concept of one function becoming proportional to another as it grows, we introduce some terminology and a new notation. In the previous example, the function $0.01n^2 + 10n$ is said to be "on the order of" the function n^2 because, as n becomes large, it becomes more nearly proportional to n^2.

To be precise, given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is ***on the order of*** $g(n)$ or that $f(n)$ is $O(g(n))$ if there exist positive integers a and b such that $f(n) \leq a * g(n)$ for all $n \geq b$). For example, if $f(n) = n^2 + 100n$, and $g(n) = n^2$, $f(n)$ is $O(g(n))$, since $n^2 + 100n$ is less than or equal to $2n^2$ for all n greater than or equal to 100. In this case, a equals 2 and b equals 100. This same $f(n)$ is also $O(n^3)$, since $n^2 + 100n$ is less than or equal to $2n^3$ for all n greater than or equal to 8. Given a function $f(n)$, there may be many functions $g(n)$ such that $f(n)$ is $O(g(n))$.

If $f(n)$ is $O(g(n))$, then "eventually" (i.e., for $n \geq b$) $f(n)$ becomes permanently smaller or equal to some multiple of $g(n)$. In a sense, we are saying that $f(n)$ is bounded by $g(n)$ from above, or that $f(n)$ is a "smaller" function than $g(n)$. Another formal

way of saying this is that $f(n)$ is ***asymptotically bounded*** by $g(n)$. Yet another interpretation is that $f(n)$ grows more slowly than $g(n)$, since, proportionately (i.e., up to a factor of a), $g(n)$ eventually becomes larger.

It is easy to show that if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$. For example, $n^2 + 100n$ is $O(n^2)$ and n^2 is $O(n^3)$ (to see this, set a and b both equal to 1); consequently $n^2 + 100n$ is $O(n^3)$. This is called the ***transitive property***.

Note that if $f(n)$ is a constant function [i.e., $f(n) = c$ for all n], then $f(n)$ is $O(1)$, since, setting a to c and b to 1, we have $c \leq c * 1$ for all $n \geq 1$. (In fact, the value of b or n is irrelevant, since a constant function's value is independent of n.)

It is also easy to show that the function $c * n$ is $O(n^k)$ for any constants c and k. To see this, simply note that $c * n$ is less than or equal to $c * n^k$ for any $n \geq 1$ (i.e., set $a = c$ and $b = 1$). It is also obvious that n^k is $O(n^{k+j})$ for any $j \geq 0$ (use $a = 1, b = 1$). We can also show that if $f(n)$ and $g(n)$ are both $O(h(n))$, then the new function $f(n) + g(n)$ is also $O(h(n))$. All of these facts together can be used to show that if $f(n)$ is any polynomial whose leading power is k [i.e., $f(n) = c_1 * n^k + c_2 * n^{k-1} + \ldots + c_k * n + c_{k+1}$], $f(n)$ is $O(n^k)$. Indeed, $f(n)$ is $O(n^{k+j})$ for any $j \geq 0$.

Although a function may be asymptotically bounded by many other functions [e.g., $10n^2 + 37n + 153$ is $O(n^2)$, $O(10n^2)$, $O(37n^2 + 10n)$, and $O(0.05n^3)$], we usually look for an asymptotic bound that is a single term with a leading coefficient of 1 and is as "close a fit" as possible. Thus we would say that $10n^2 + 37n + 153$ is $O(n^2)$, although it is also asymptotically bounded by many other functions. Ideally, we would like to find a function $g(n)$ such that $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. If $f(n)$ is a constant or a polynomial, this can always be done by using its highest term with a coefficient of 1. For more complex functions, however, it is not always possible to find such a tight fit.

An important function in the study of algorithm efficiency is the logarithm function. Recall that $\log_m n$ is the value x such that m^x equals n. m is called the ***base*** of the logarithm. Consider the functions $\log_m n$ and $\log_k n$. Let xm be $\log_m n$ and xk be $\log_k n$. Then

$$m^{xm} = n \text{ and } k^{xk} = n$$

so that

$$m^{xm} = k^{xk}$$

Taking $\log_m$ of both sides,

$$xm = \log_m(k^{xk})$$

Now it can easily be shown than $\log_z(x^y)$ equals $y * \log_z x$ for any x, y, and z, so that the last equation can be rewritten as (recall that $xm = \log_m n$)

$$\log_m n = xk * \log_m k$$

or as (recall that $xk = \log_k n$)

$$\log_m n = (\log_m k) * \log_k n$$

Thus $\log_m n$ and $\log_k n$ are constant multiples of each other.

It is easy to show that if $f(n) = c * g(n)$, where c is a constant, then $f(n)$ is $O(g(n))$ [indeed, we have already shown that this is true for the method $f(n) = n^k$].

Thus $\log_m n$ is $O(\log_k n)$ and $\log_k n$ is $O(\log_k \mathrm{n})$ for any m and k. Since each logarithm function is on the order of any other, we usually omit the base when speaking of methods of logarithmic order and say that all such methods are $O(\log n)$.

The following facts establish an order hierarchy of methods:

c is $O(1)$ for any constant c.
c is $O(\log n)$, but $\log n$ is not $O(1)$.
$c * \log_k n$ is $O(\log n)$ for any constants c, k.
$c * \log_k n$ is $O(n)$, but n is not $O(\log n)$.
$c * n^k$ is $O(n^k)$ for any constants c, k.
$c * n^k$ is $O(n^{k+j})$, but n^{k+j} is not $O(n^k)$.
$c * n * \log_k n$ is $O(n \log n)$ for any constants c, k.
$c * n * \log_k n$ is $O(n^2)$, but n^2 is not $O(n \log n)$.
$c * n^j * \log_k n$ is $O(n^j \log n)$ for any constants c, j, k.
$c * n^j * \log_k n$ is $O(n^{j+1})$, but n^{j+1} is not $O(n^j \log n)$.
$c * n^j * (\log_k n)^l$ is $O(n^j(\log_k n)^l)$ for any constants c, j, k, l.
$c * n^j * (\log_k n)^l$ is $O(n^{j+1})$, but n^{j+1} is not $(n^j(\log n)^l)$.
$c * n^j * (\log_k n)^l$ is $O(n^j(\log n)^{l+1})$, but $n^j(\log_k n)^{j+1}$ is not $O(n^j(\log n)^l)$.
$c * n^k$ is $O(d^n)$, but d^n is not $O(n^k)$ for any constants c and k, and $d > 1$.

The hierarchy of functions established by these facts, with each method of lower order than the next, is c, $\log n$, $(\log n)^k$, n, $n(\log n)^k$, n^k, $n^k(\log n)^l$, n^{k+1}, d^n.

Functions that are $O(n^k)$ for some k are said to be of ***polynomial*** order, while functions that are $O(d^n)$ for some $d > 1$ but not $O(n^k)$ for any k are said to be of ***exponential order***.

The distinction between polynomial-order functions and exponential-order functions is extremely important. Even a small exponential-order function, such as 2^n, grows far larger than any polynomial-order function, such as n^k regardless of the size of k. As an illustration of the rapidity with which exponential-order functions grow, consider that 2^{10} equals 1024 but 2^{100} (i.e., 1024^{10}) is greater than the number formed by a 1 followed by thirty zeros. The smallest k for which 10^k exceeds 2^{10} is 4, but the smallest k for which 100^k exceeds 2^{100} is 16. As n becomes larger, larger values of k are needed for n to keep up with 2^n. For any single k, 2^n eventually becomes permanently larger than n^k.

Because of the incredible rate of growth of exponential-order functions, problems that require exponential-time algorithms for solution are considered to be ***intractable*** on current computing equipment. That is, such problems cannot be practically solved except in the simplest cases.

Efficiency of Sorting

Using the concept of the order of a sort, we can compare various sorting techniques and classify them, in general terms, as "good" or "bad". One might hope to discover the "optimal" sort which is $O(n)$ regardless of the contents or order of the input. Unfortunately, however, it can be shown that no such generally useful sort exists. Most of the classical sorts we shall consider have time requirements that range from $O(n \log n)$ to

n	$n \log_{10} n$	n^2
1×10^1	1.0×10^1	1.0×10^2
5×10^1	8.5×10^1	2.5×10^3
1×10^2	2.0×10^2	1.0×10^4
5×10^2	1.3×10^3	2.5×10^5
1×10^3	3.0×10^3	1.0×10^6
5×10^3	1.8×10^4	2.5×10^7
1×10^4	4.0×10^4	1.0×10^8
5×10^4	2.3×10^5	2.5×10^9
1×10^5	5.0×10^5	1.0×10^{10}
5×10^5	2.8×10^6	2.5×10^{11}
1×10^6	6.0×10^6	1.0×10^{12}
5×10^6	3.3×10^7	2.5×10^{13}
1×10^7	7.0×10^7	1.0×10^{14}

FIGURE 6.1.4 Comparison of $n \log n$ and n^2 for various values of n.

$O(n^2)$. In the former, multiplying the file size by 100 will multiply the sorting time by less than 200; in the latter, multiplying the file size by 100 multiplies the sorting time by a factor of 10,000. Figure 6.1.4 shows the comparison of $n \log n$ with n^2 for a range of values of n. It can be seen from the figure that for large n, as n increases, n^2 increases at a much more rapid rate than $n \log n$. However, a sort should not be selected simply because it is $O(n \log n)$. The relation of the file size n and the other terms comprising the actual sorting time must be known. Terms that play an insignificant role for large values of n may play a very dominant role for small values of n. All of these issues must be considered before an intelligent sort selection can be made.

A second method of determining the time requirements of a sorting technique is to actually run the program and measure its efficiency (either by measuring absolute time units or by the number of operations performed). In order for such results to be used in measuring the efficiency of a sort, the test must be run on "many" sample files. Even when such statistics have been gathered, the application of the sort to a specific file may not yield results that follow the general pattern. Peculiar attributes of the file in question may make the sorting speed deviate significantly. In the sorts in the subsequent sections, we shall give an intuitive explanation as to why a particular sort is classified as $O(n^2)$ or $O(n \log n)$; we leave mathematical analysis and sophisticated testing of empirical data as exercises for the ambitious reader.

In most cases, the time needed by a sort depends upon the original sequence of the data. For some sorts, input data that is almost in sorted order can be completely sorted in time $O(n)$, while input data that is in reverse order needs time which is $O(n^2)$. For other sorts, the time required is $O(n \log n)$ regardless of the original order of the data. Thus if we have some knowledge about the original sequence of the data, we can make a more intelligent decision as to which sorting method to select. On the other hand, if we have no such knowledge, we may wish to select a sort based either on the worst possible case or on the "average" case. In any event, the only general comment that can be made about sorting techniques is that there is no "best" general sorting technique. The choice of a sort must, of necessity, depend on the specific circumstances.

Once a sorting technique has been selected, the programmer should make the program as efficient as possible. In many programming applications it is necessary to

sacrifice efficiency for the sake of clarity. With sorting, the situation is usually the opposite. Once a sorting program has been written and tested, the programmer's chief goal is to improve its speed, even if it becomes less readable. The reason for this is that a sort may account for the major part of a program's efficiency, so that any improvement in sorting time significantly affects overall efficiency. Another reason is that a sort is often used quite frequently, so that a small improvement in its execution speed saves a great deal of computer time. It is usually a good idea to remove method calls, especially from inner loops, and replace them with the code of the method in line, since the call-return mechanism of a language can be prohibitively expensive in terms of time. Also, a method call may involve the assignment of storage to local variables, an activity that sometimes requires a call to the operating system. In many programs we do not do this so as not to obfuscate the intent of the program with huge blocks of code.

Space constraints are usually less important than time considerations. One reason is that the amount of space needed for most sorting programs is closer to $O(n)$ than to $O(n^2)$. A second reason is that if more space is required, it can almost always be found in auxiliary storage. An ideal sort is an ***in-place sort*** whose additional space requirements are $O(1)$. That is, an in-place sort manipulates the elements to be sorted within the array or list space that contained the original unsorted input. The additional space that is required is in the form of a constant number of locations (e.g., declared individual program variables) regardless of the size of the set to be sorted.

The expected relationship between time and space usually holds for sorting algorithms: programs that require less time usually require more space, and vice versa. However, there are clever algorithms that utilize both minimum time and minimum space; that is, they are $O(n \log n)$ in-place sorts. These may, however, require more programmer time to develop and verify. They also have higher constants of proportionality than many sorts that use more space or have higher time-orders and so require more time to sort small sets.

In the remaining sections we investigate some of the more popular sorting techniques and indicate some of their advantages and disadvantages.

EXERCISES

6.1.1 Choose any sorting technique with which you are familiar.

a. Write a program for the sort.

b. Is the sort stable?

c. Determine the time requirements of the sort as a method of the file size, both mathematically and empirically.

d. What is the order of the sort?

e. At what file size does the most dominant term begin to overshadow the others?

6.1.2 Show that the method $(\log_m n)^k$ is $O(n)$ for all m and k, but that n is not $O((\log n)^k)$ for any k.

6.1.3 Suppose a time requirement is given by the formula $a * n^2 + b * n * \log_2 n$, where a and b are constants. Answer the following questions by both proving your results mathematically and writing a program to validate the results empirically.

a. For what values of n (expressed in terms of a and b) does the first term dominate the second?
b. For what value of n (expressed in terms of a and b) are the two terms equal?
c. For what values of n (expressed in terms of a and b) does the second term dominate the first?

6.1.4 Show that any process that sorts a file can be extended to find all the duplicates in the file.

6.1.5 A ***sort decision tree*** is a binary tree that represents a sorting method based on comparisons. Figure 6.1.5 illustrates such a decision tree for a file of three elements. Each nonleaf of such a tree represents a comparison between two elements. Each leaf represents a completely sorted file. A left branch from a nonleaf indicates that the first key was smaller than the second; a right branch indicates that it was larger. (We assume that all the elements in the file have distinct keys.) For example, the tree in Figure 6.1.5 represents a sort on three elements $x[0], x[1], x[2]$ that proceeds as follows:

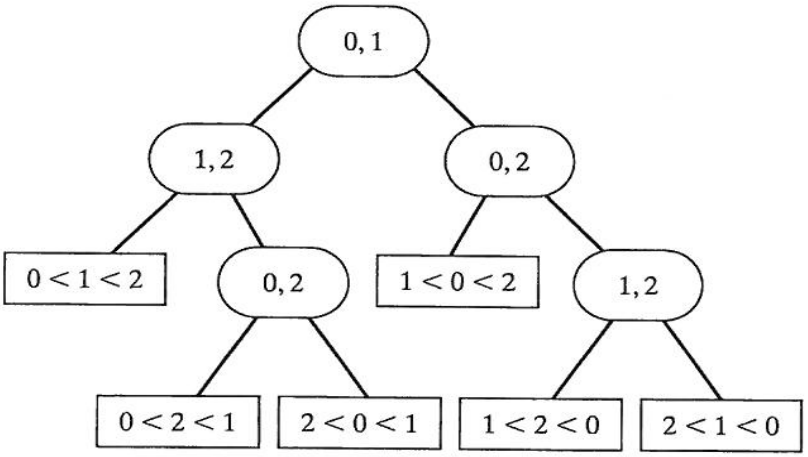

FIGURE 6.1.5 Decision tree for a file of three elements.

Compare $x[0]$ to $x[1]$. If $x[0] < x[1]$, then compare $x[1]$ with $x[2]$, and if $x[1] < x[2]$, then the sorted order of the file is $x[0]$, $x[1]$, $x[2]$; otherwise if $x[0] < x[2]$, the sorted order is $x[0], x[2], x[1]$, and if $x[0] > x[2]$, then the sorted order is $x[2], x[0], x[1]$. If $x[0] > x[1]$, then proceed in a similar fashion down the right subtree.

a. Show that a sort decision tree that never makes a redundant comparison (i.e., never compares $x[i]$ and $x[j]$ if the relationship between $x[i]$ and $x[j]$ is known) has $n!$ leafs.
b. Show that the depth of such a decision tree is at least $\log_2(n!)$.
c. Show that $n! \geq (n/2)^{n/2}$ so that the depth of such a tree is $O(n \log n)$.
d. Explain why this proves that any sorting method that uses comparisons on a file of size n must make at least $O(n \log n)$ comparisons.

6.1.6 Given a sort decision tree for a file, as in the previous exercise, show that if the file contains some equal elements, the result of applying the tree to the file (where either a left or right branch is taken whenever two elements are equal) is a sorted file.