CHAPTER SEVEN

Quicksort

THE SUBJECT OF this chapter is the sorting algorithm that is probably used more widely than any other, *quicksort*. The basic algorithm was invented in 1960 by C. A. R. Hoare, and it has been studied by many people since that time (*see reference section*). Quicksort is popular because it is not difficult to implement, works well for a variety of different kinds of input data, and consumes fewer resources than any other sorting method in many situations.

The quicksort algorithm has the desirable features that it is in-place (uses only a small auxiliary stack), requires requires time only proportional to $N \log N$ on the average to sort N items, and has an extremely short inner loop. Its drawbacks are that it is not stable, takes about N^2 operations in the worst case, and is fragile in the sense that a simple mistake in the implementation can go unnoticed and can cause it to perform badly for some files.

The performance of quicksort is well understood. The algorithm has been subjected to a thorough mathematical analysis, and we can make precise statements about its performance. The analysis has been verified by extensive empirical experience, and the algorithm has been refined to the point where it is the method of choice in a broad variety of practical sorting applications. It is therefore worthwhile for us to look more carefully than for other algorithms at ways of implementing quicksort efficiently. Similar implementation techniques are appropriate for other algorithms; with quicksort, we can use them with confidence, because we know precisely how they will affect performance.

It is tempting to try to develop ways to improve quicksort: A faster sorting algorithm is computer science's "better mousetrap," and quicksort is a venerable method that seems to invite tinkering. Almost from the moment Hoare first published the algorithm, improved versions have been appearing in the literature. Many ideas have been tried and analyzed, but it is easy to be deceived, because the algorithm is so well balanced that the effects of improvements in one part of the program can be more than offset by the effects of bad performance in another part of the program. We examine in detail three modifications that do improve quicksort substantially.

A carefully tuned version of quicksort is likely to run significantly faster on most computers than will any other sorting method, and quicksort is widely used as a library sort utility and for other serious sorting applications. Indeed, the standard C library's sort is called `qsort`, since quicksort is typically the underlying algorithm used in implementations. However, the running time of quicksort depends on the input, ranging from linear to quadratic in the number of items to be sorted, and people are sometimes surprised by undesirable and unexpected effects for some inputs, particularly in highly tuned versions of the algorithm. If an application does not justify the work required to be sure that a quicksort implementation is not flawed, shellsort might well be a safer choice that will perform well for less implementation investment. For huge files, however, quicksort is likely to run five to ten times as fast as shellsort, and it can adapt to be even more efficient for other types of files that might occur in practice.

7.1 The Basic Algorithm

Quicksort is a divide-and-conquer method for sorting. It works by *partitioning* an array into two parts, then sorting the parts independently. As we shall see, the precise position of the partition depends on the initial order of the elements in the input file. The crux of the method is the partitioning process, which rearranges the array to make the following three conditions hold:

- The element `a[i]` is in its final place in the array for some `i`.
- None of the elements in `a[l]`, . . . , `a[i-1]` is greater than `a[i]`.
- None of the elements in `a[i+1]`, . . . , `a[r]` is less than `a[i]`.

Program 7.1 Quicksort

If the array has one or fewer elements, do nothing. Otherwise, the array is processed by a `partition` procedure (see Program 7.2), which puts `a[i]` into position for some `i` between `l` and `r` inclusive, and rearranges the other elements such that the recursive calls properly finish the sort.

```
 int partition(Item a[], int l, int r);
void quicksort(Item a[], int l, int r)
  { int i;
    if (r <= l) return;
    i = partition(a, l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
```

We achieve a complete sort by partitioning, then recursively applying the method to the subfiles, as depicted in Figure 7.1. Because the partitioning process always puts at least one element into position, a formal proof by induction that the recursive method constitutes a proper sort is not difficult to develop. Program 7.1 is a recursive program that implements this idea.

We use the following general strategy to implement partitioning. First, we arbitrarily choose `a[r]` to be the *partitioning element*—the one that will go into its final position. Next, we scan from the left end of the array until we find an element greater than the partitioning element, and we scan from the right end of the array until we find an element less than the partitioning element. The two elements that stopped the scans are obviously out of place in the final partitioned array, so we exchange them. Continuing in this way, we ensure that no array elements to the left of the left pointer are greater than the partitioning element, and no array elements to the right of the right pointer are less than the partitioning element, as depicted in the following diagram:

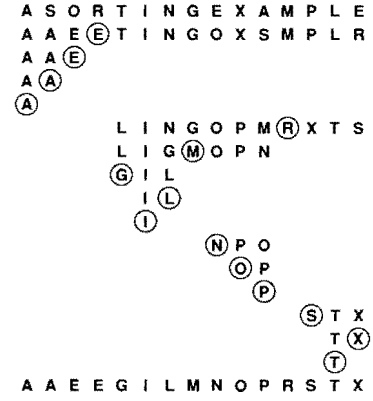


Figure 7.1
Quicksort example

Quicksort is a recursive partitioning process: We partition a file by putting some element (the partitioning element) in place, and rearranging the array such that smaller elements are to the left of the partitioning element and larger elements to its right. Then, we sort the left and right parts of the array recursively. Each line in this diagram depicts the result of partitioning the displayed subfile using the circled element. The end result is a fully sorted file.

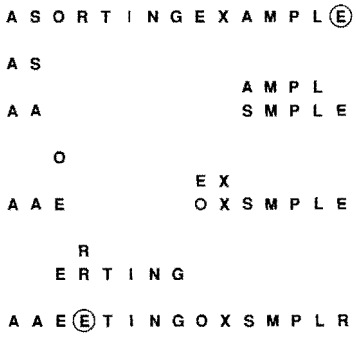


Figure 7.2
Quicksort partitioning

Quicksort partitioning begins with the (arbitrary) choice of a partitioning element. Program 7.2 uses the rightmost element E. *Then, it scans from the left over smaller elements and from the right over larger elements, exchanges the elements that stop the scans, continuing until the scan pointers meet. First, we scan from the left and stop at the* S, *then we scan from the right and stop at the* A, *and then we exchange the* S *and the* A. *Next, we continue the scan from the left until we stop at the* O, *and continue the scan from the right until we stop at the* E, *then exchange the* O *and the* E. *Next, our scanning pointers cross: We continue the scan from the left until we stop at the* R, *then continue the scan from the right (past the* R*) until we stop at the* E. *To finish the process, we exchange the partitioning element (the* E *at the right) with the* R.

Here, `v` refers to the value of the partitioning element, `i` to the left pointer, and `j` to the right pointer. As indicated in this diagram, it is best to stop the left scan for elements greater than *or equal to* the partitioning element and the right scan for elements less than *or equal to* the partitioning element, even though this policy might seem to create unnecessary exchanges involving elements equal to the partitioning element (we shall examine the reasons for this policy later in this section). When the scan pointers cross, all that we need to do to complete the partitioning process is to exchange `a[r]` with the leftmost element of the right subfile (the element pointed to by the left pointer). Program 7.2 is an implementation of this process, and Figures 7.2 and 7.3 depict examples.

The inner loop of quicksort increments a pointer and compares an array element against a fixed value. This simplicity is what makes quicksort quick: It is hard to envision a shorter inner loop in a sorting algorithm.

Program 7.2 uses an explicit test to stop the scan if the partitioning element is the smallest element in the array. It might be worthwhile to use a sentinel to avoid this test: The inner loop of quicksort is so small that this one superfluous test could have a noticeable effect on performance. A sentinel is not needed for this implementation when the partitioning element is the largest element in the file, because the partitioning element itself is at the right end of the array to stop the scan. Other implementations of partitioning discussed later in this section and elsewhere in this chapter do not necessarily stop the scan on keys equal to the partitioning element—we might need to add a test to stop the pointer from running off the right end of the array in such an implementation. On the other hand, the improvement to quicksort that we discuss in Section 7.5 has the side benefit of needing neither the test nor a sentinel at either end.

The partitioning process is not stable, because any key might be moved past a large number of keys equal to it (which have not even been examined yet) during any exchange. No easy way to make an array-based quicksort stable is known.

The partitioning procedure must be implemented carefully. Specifically, the most straightforward way to guarantee that the recursive program terminates is that it (*i*) does not call itself for files of size 1 or less; and (*ii*) calls itself for *only* files that are strictly smaller

Program 7.2 Partitioning

The variable `v` holds the value of the partitioning element `a[r]`, and `i` and `j` are the left and right scan pointers, respectively. The partitioning loop increments `i` and decrements `j`, while maintaining the invariant property that no elements to the left of `i` are greater than `v` and no elements to the right of `j` are smaller than `v`. Once the pointers meet, we complete the partitioning by exchanging `a[i]` and `a[r]`, which puts `v` into `a[i]`, with no larger elements to v's right and no smaller elements to its left.

The partitioning loop is implemented as an infinite loop, with a `break` when the pointers cross. The test `j == l` protects against the case that the partitioning element is the smallest element in the file.

```
int partition(Item a[], int l, int r)
  { int i = l-1, j = r; Item v = a[r];
    for (;;)
      {
        while (less(a[++i], v)) ;
        while (less(v, a[--j])) if (j == l) break;
        if (i >= j) break;
        exch(a[i], a[j]);
      }
    exch(a[i], a[r]);
    return i;
  }
```

than given as input. These policies may seem obvious, but it is easy to overlook a property of the input that can lead to a spectacular failure. For instance, a common mistake in implementing quicksort is not ensuring that one element is always put into position, then falling into an infinite recursive loop when the partitioning element happens to be the largest or smallest element in the file.

When duplicate keys are present in the file, the pointer crossing is subtle. We could improve the partitioning process slightly by terminating the scans when `j < i`, and then using `j`, rather than `i-1`, to delimit the right end of the left subfile for the first recursive call. Letting the loop iterate one more time in this case is an improvement, because, whenever the scanning loops terminate with `j` and `i` referring to the same element, we end up with *two* elements in their final posi-

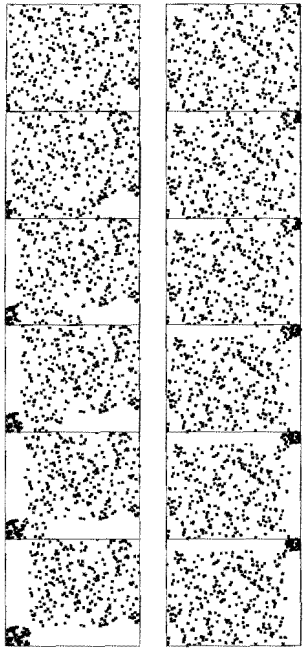


Figure 7.3
Dynamic characteristics of quicksort partitioning

The partitioning process divides a file into two subfiles that can be sorted independently. None of the elements to the left of the left scan pointer is larger, so there are no dots above and to its left; and none of the elements to the right of the right scan pointer is smaller, so there are no dots below and to its right. As shown in these two examples, partitioning a random array divides it into two smaller random arrays, with one element (the partitioning element) ending up on the diagonal.

tions: the element that stopped both scans, which must therefore be equal to the partitioning element, and the partitioning element itself. This case would occur, for example, if R were E in Figure 7.2. This change is probably worth making, because, in this particular case, the program as given leaves a record with a key equal to the partitioning key in `a[r]`, and that makes the first partition in the call `quicksort(a, i+1, r)` degenerate, because its rightmost key is its smallest. Separating partitioning out as in Programs 7.1 and 7.2 makes quicksort a bit easier to understand, however, so we refer to the combination as the basic quicksort algorithm. If significant numbers of duplicate keys might be present, other factors come into play. We consider them next.

There are three basic strategies that we could adopt with respect to keys equal to the partitioning element: have both pointers stop on such keys (as in Program 7.2); have one pointer stop and the other scan over them; or have both pointers scan over them. The question of which of these strategies is best has been studied in detail mathematically, and results show that it is best to have both pointers stop, primarily because this strategy tends to balance the partitions in the presence of many duplicate keys, whereas the other two can lead to badly unbalanced partitions for some files. We also consider a slightly more complicated and much more effective method for dealing with duplicate keys in Section 7.6.

Ultimately, the efficiency of the sort depends on how well the partitioning divides the file, which in turn depends on the value of the partitioning element. Figure 7.2 shows that partitioning divides a large randomly ordered file into two smaller randomly ordered files, but that the actual split could be anywhere in the file. We would prefer to choose an element that would split the file near the middle, but we do not have the necessary information to do so. If the file is randomly ordered, choosing `a[r]` as the partitioning element is the same as choosing any other specific element, and will give us a split near the middle *on the average*. In Section 7.4 we consider the analysis of the algorithm that allows us to see how this choice compares to the ideal choice. In Section 7.5 we see how the analysis guides us in considering choices of the partitioning element that make the algorithm more efficient.

Exercises

▷ 7.1 Show, in the style of the example given here, how quicksort sorts the file E A S Y Q U E S T I O N.

7.2 Show how the file 1 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 is partitioned, using both Program 7.2 and the minor modifications suggested in the text.

7.3 Implement partitioning without using a `break` statement or a `goto` statement.

• 7.4 Develop a stable quicksort for linked lists.

○ 7.5 What is the maximum number of times during the execution of quicksort that the largest element can be moved, for a file of N elements?

7.2 Performance Characteristics of Quicksort

Despite its many assets, the basic quicksort program has the definite liability that it is extremely inefficient on some simple files that can arise in practice. For example, if it is called with a file of size N that is already sorted, then all the partitions will be degenerate, and the program will call itself N times, removing just one element for each call.

Property 7.1 *Quicksort uses about $N^2/2$ comparisons in the worst case.*

By the argument just given, the number of comparisons used for a file that is already in order is

$$N + (N-1) + (N-2) + \ldots + 2 + 1 = (N+1)N/2.$$

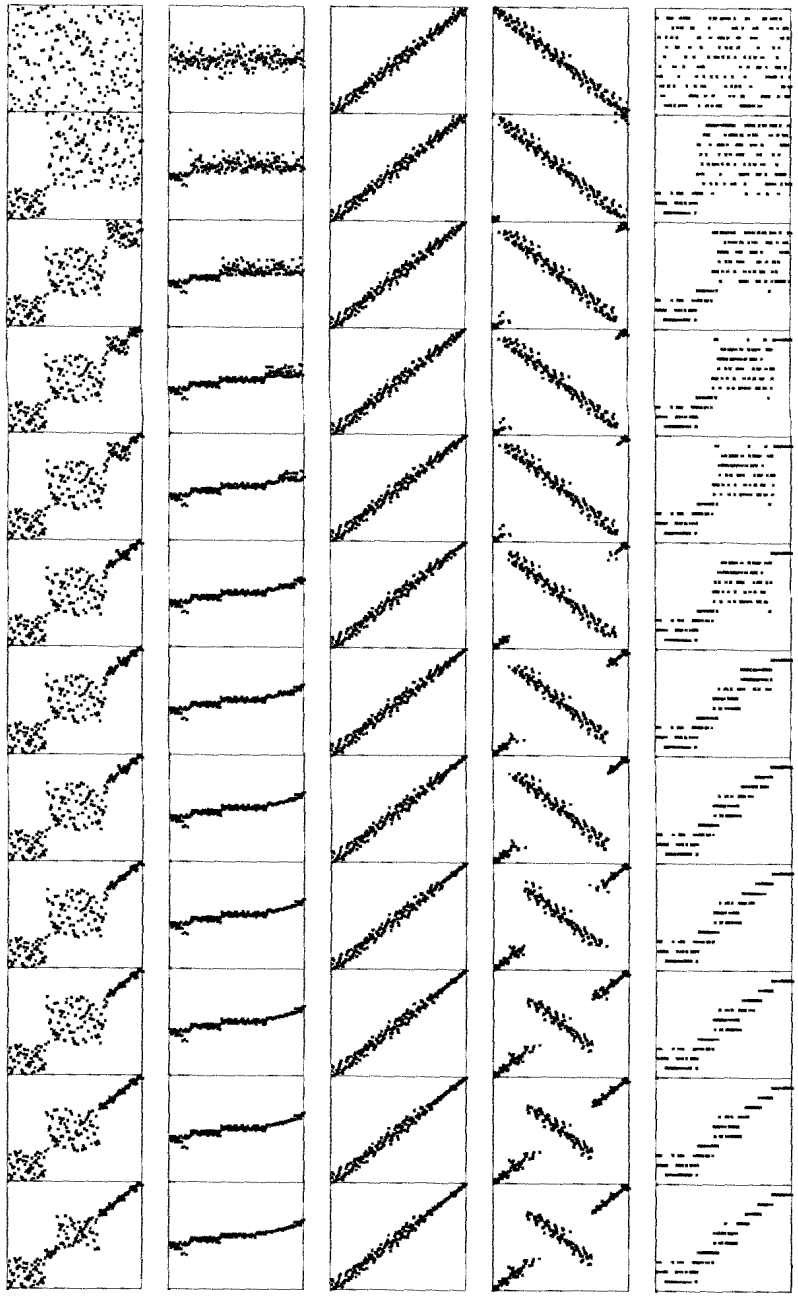
All the partitions are also degenerate for files in reverse order, as well as for other kinds of files that are less likely to occur in practice (see Exercise 7.6). ■

This behavior means not only that the time required will be about $N^2/2$, but also that the space required to handle the recursion will be about N (see Section 7.3), which is unacceptable for large files. Fortunately, there are relatively easy ways to reduce drastically the likelihood that this worst case will occur in typical applications of the program.

The best case for quicksort is when each partitioning stage divides the file exactly in half. This circumstance would make the number of

Figure 7.4
Dynamic characteristics of quicksort on various types of files

The choice of an arbitrary partitioning element in quicksort results in differing partitioning scenarios for different files. These diagrams illustrate the initial portions of scenarios for files that are random, Gaussian, nearly ordered, nearly reverse ordered, and randomly ordered with 10 distinct key values (left to right), *using a relatively large value of the cutoff for small subfiles. Elements not involved in partitioning end up close to the diagonal, leaving an array that could be handled easily by insertion sort. The nearly ordered files require an excessive number of partitions.*



comparisons used by quicksort satisfy the divide-and-conquer recurrence

$$C_N = 2C_{N/2} + N.$$

The $2C_{N/2}$ covers the cost of sorting the two subfiles; the N is the cost of examining each element, using one partitioning pointer or the other. From Chapter 5, we know that this recurrence has the solution

$$C_N \approx N \lg N.$$

Although things do not always go this well, it is true that the partition falls in the middle *on the average*. Taking into account the precise probability of each partition position makes the recurrence more complicated and more difficult to solve, but the final result is similar.

Property 7.2 *Quicksort uses about* $2N \ln N$ *comparisons on the average.*

The precise recurrence formula for the number of comparisons used by quicksort for N randomly ordered distinct elements is

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \le k \le N} (C_{k-1} + C_{N-k}) \quad \text{for } N \ge 2,$$

with $C_1 = C_0 = 0$. The $N + 1$ term covers the cost of comparing the partitioning element with each of the others (two extra for where the pointers cross); the rest comes from the observation that each element k is likely to be the partitioning element with probability $1/k$, after which we are left with random files of size $k - 1$ and $N - k$.

Although it looks rather complicated, this recurrence is actually easy to solve, in three steps. First, $C_0 + C_1 + \cdots + C_{N-1}$ is the same as $C_{N-1} + C_{N-2} + \cdots + C_0$, so we have

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \le k \le N} C_{k-1}.$$

Second, we can eliminate the sum by multiplying both sides by N and subtracting the same formula for $N - 1$:

$$NC_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}.$$

This formula simplifies to the recurrence

$$NC_N = (N+1)C_{N-1} + 2N.$$

Third, dividing both sides by $N(N+1)$ gives a recurrence that telescopes:

$$\begin{aligned}\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \vdots \\ &= \frac{C_2}{3} + \sum_{3\le k\le N} \frac{2}{k+1}.\end{aligned}$$

This exact answer is nearly equal to a sum that is easily approximated by an integral (see Section 2.3):

$$\frac{C_N}{N+1} \approx 2 \sum_{1\le k<N} \frac{1}{k} \approx 2 \int_1^N \frac{1}{x} dx = 2 \ln N,$$

which implies the stated result. Note that $2N \ln N \approx 1.39 N \lg N$, so the average number of comparisons is only about 39 percent higher than in the best case. ■

This analysis assumes that the file to be sorted comprises randomly ordered records with distinct keys, but the implementation in Programs 7.1 and 7.2 can run slowly in some cases when the keys are not necessarily distinct and not necessarily in random order, as illustrated in Figure 7.4. If the sort is to be used a great many times or if it is to be used to sort a huge file (or, in particular, if it is to be used as a general-purpose library sort that will be used to sort files of unknown characteristics), then we need to consider several of the improvements discussed in Sections 7.5 and 7.6 that can make it much less likely that a bad case will occur in practice, while also reducing the average running time by 20 percent.

Exercises

7.6 Give six files of 10 elements for which quicksort (Program 7.1) uses the same number of comparisons as the worst-case file (when all the elements are in order).

7.7 Write a program to compute the exact value of C_N, and compare the exact value with the approximation $2N \ln N$ for $N = 1000$, 10000, and 100000.

○ 7.8 About how many comparisons will quicksort (Program 7.1) make when sorting a file of N equal elements?

7.9 About how many comparisons will quicksort (Program 7.1) make when sorting a file consisting of N items that have just two different key values (k items with one value, $N - k$ items with the other)?

• 7.10 Write a program that produces a best-case file for quicksort: a file of N distinct elements with the property that every partition will produce subfiles that differ in size by at most 1.

7.3 Stack Size

As we did in Chapter 3, we can use an explicit pushdown stack for quicksort, thinking of the stack as containing work to be done in the form of subfiles to be sorted. Any time that we need a subfile to process, we pop the stack. When we partition, we create two subfiles to be processed and push both on the stack. In the recursive implementation in Program 7.1, the stack maintained by the system holds this same information.

For a random file, the maximum size of the stack is proportional to $\log N$ (*see reference section*), but the stack can grow to size proportional to N for a degenerate case, as illustrated in Figure 7.5. Indeed, the very worst case is when the input file is already sorted. The potential for stack growth proportional to the size of the original file is a subtle but real difficulty with a recursive implementation of quicksort: There is always an underlying stack, and a degenerate case on a large file could cause the program to terminate abnormally because of lack of memory—behavior obviously undesirable for a library sorting routine. (Actually, we likely would run out of time before running out of space.) It is difficult to provide a *guarantee* against this behavior, but we shall see in Section 7.5 that it is not difficult to provide safeguards that make such degenerate cases extremely unlikely to occur.

Program 7.3 is a nonrecursive implementation that addresses this problem by checking the sizes of the two subfiles and putting the larger of the two on the stack first. Figure 7.6 illustrates this policy. Comparing this example with Figure 7.1, we see that the subfiles are not changed by this policy; only the order in which they are processed is changed. Thus, we save on space costs without affecting time costs.

The policy of putting the larger of the small subfiles on the stack ensures that each entry on the stack is no more than one-half of the size of the one below it, so that the stack needs to contain room for

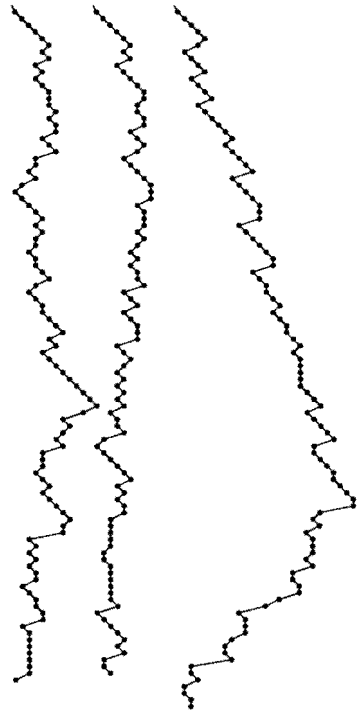


Figure 7.5
Stack size for quicksort

The recursive stack for quicksort does not grow large for random files, but can take excessive space for degenerate files. The stack sizes for two random files (left, center) *and that for a partially ordered file* (right) *are plotted here.*

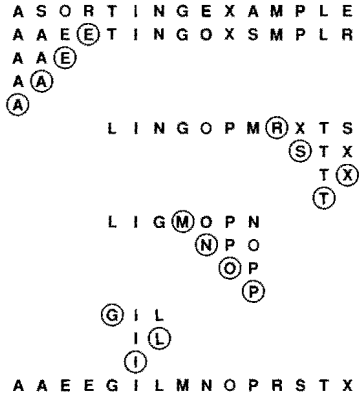


Figure 7.6
Quicksort example (sorting the smaller subfile first)

The order in which the subfiles are processed does not affect the correct operation of the quicksort algorithm, or the time taken, but might affect the size of the pushdown stack underlying the recursive structure. Here the smaller of the two subfiles is processed first after each partition.

Program 7.3 Nonrecursive quicksort

This nonrecursive implementation (see Chapter 5) of quicksort uses an explicit pushdown stack, replacing recursive calls with stack pushes (of the parameters) and the procedure call/exit with a loop that pops parameters from the stack and processes them as long as the stack is nonempty. We put the larger of the two subfiles on the stack first to ensure that the maximum stack depth for sorting N elements is $\lg N$ (see Property 7.3).

```
#define push2(A, B)  push(B); push(A);
void quicksort(Item a[], int l, int r)
  { int i;
    stackinit(); push2(l, r);
    while (!stackempty())
      {
        l = pop(); r = pop();
        if (r <= l) continue;
        i = partition(a, l, r);
        if (i-l > r-i)
          { push2(l, i-1); push2(i+1, r); }
        else
          { push2(i+1, r); push2(l, i-1); }
      }
  }
```

only about $\lg N$ entries. This maximum stack usage occurs when the partition always falls at the center of the file. For random files, the actual maximum stack size is much lower; for degenerate files it is likely to be small.

Property 7.3 *If the smaller of the two subfiles is sorted first, then the stack never has more than* $\lg N$ *entries when quicksort is used to sort* N *elements.*

The worst-case stack size must be less than T_N, where T_N satisfies the recurrence $T_N = T_{\lfloor N/2 \rfloor} + 1$ with $T_1 = T_0 = 0$. This recurrence is a standard one of the type considered in Chapter 5 (see Exercise 7.13). ■

This technique does not necessarily work in a truly recursive implementation, because it depends on *end-* or *tail-recursion removal.* If the last action of a procedure is to call another procedure, some programming environments will arrange things such that local variables are cleared from the stack *before*, rather than after, the call. Without end-recursion removal, we cannot guarantee that the stack size will be small for quicksort. For example, a call to quicksort for a file of size N that is already sorted will result in a recursive call to such a file of size $N - 1$, in turn resulting in a recursive call for such a file of size $N - 2$, and so on, ultimately resulting in a stack depth proportional to N. This observation would seem to suggest using a nonrecursive implementation to guard against excessive stack growth. On the other hand, some C compilers automatically remove end recursion, and many machines have direct hardware support for function calls—the nonrecursive implementation in Program 7.3 might therefore actually be slower than the recursive implementation in Program 7.1 in such environments.

Figure 7.7 further illustrates the point that the nonrecursive method processes the same subfiles (in a different order) as does the recursive method for any file. It shows a tree structure with the partitioning element at the root and the trees corresponding to the left and right subfiles as left and right children, respectively. Using the recursive implementation of quicksort corresponds to visiting the nodes of this tree in preorder; the nonrecursive implementation corresponds to a visit-the-smaller-subtree-first traversal rule.

When we use an explicit stack, as we did in Program 7.3, we avoid some of the overhead implicit in a recursive implementation, although modern programming systems do not incur much overhead for such simple programs. Program 7.3 can be further improved. For example, it puts both subfiles on the stack, only to have the top one immediately popped off; we could change it to set the variables `l` and `r` directly. Also, the test for `r <= l` is done as subfiles come off the stack, whereas it would be more efficient never to put such subfiles on the stack (see Exercise 7.14). This case might seem insignificant, but the recursive nature of quicksort actually ensures that a large fraction of the subfiles during the course of the sort are of size 0 or 1. Next, we examine an important improvement to quicksort that gains efficiency by expanding upon this idea, handling all small subfiles in as efficient a manner as possible.

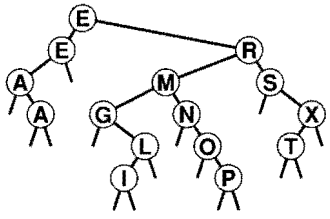


Figure 7.7
Quicksort partitioning tree

If we collapse the partitioning diagrams in Figures 7.1 and 7.6 by connecting each partitioning element to the partitioning element used in its two subfiles, we get this static representation of the partitioning process (in both cases). In this binary tree, each subfile is represented by its partitioning element (or by itself, if it is of size 1), and the subtrees of each node are the trees representing the subfiles after partitioning. For clarity, null subfiles are not shown here, although our recursive versions of the algorithm do make recursive calls with `r < l` *when the partitioning element is the smallest or largest element in the file. The tree itself does not depend on the order in which the subfiles are partitioned. Our recursive implementation of quicksort corresponds to visiting the nodes of this tree in preorder; our nonrecursive implementation corresponds to a visit-the-smaller-subtree-first rule.*

Exercises

▷7.11 Give, in the style of Figure 5.5, the stack contents after each pair of *push* and *pop* operations, when Program 7.3 is used to sort a file with the keys EASYQUESTION.

▷7.12 Answer Exercise 7.11 for the case where we always push the right subfile, then the left subfile (as is the case in the recursive implementation).

7.13 Complete the proof of Property 7.3, by induction.

7.14 Revise Program 7.3 such that it never puts on the stack subfiles with `r <= l`.

▷7.15 Give the maximum stack size required by Program 7.3 when $N = 2^n$.

7.16 Give the maximum stack sizes required by Program 7.3 when $N = 2^n - 1$ and $N = 2^n + 1$.

○7.17 Would it be reasonable to use a queue instead of a stack for a nonrecursive implementation of quicksort? Explain your answer.

7.18 Determine and report whether your programming environment implements end-recursion removal.

•7.19 Run empirical studies to determine the average stack size used by the basic recursive quicksort algorithm for random files of N elements, for $N = 10^3$, 10^4, 10^5, and 10^6.

••7.20 Find the average number of subfiles of size 0, 1, and 2 when quicksort is used to sort a random file of N elements.

7.4 Small Subfiles

A definite improvement to quicksort arises from the observation that a recursive program is guaranteed to call itself for many small subfiles, so it should use as good a method as possible when it encounters small subfiles. One obvious way to arrange for it to do so is to change the test at the beginning of the recursive routine from a `return` to a call on insertion sort, as follows:

```
if (r-l <= M) insertion(a, l, r);
```

Here, M is some parameter whose exact value depends upon the implementation. We can determine the best value for M either through analysis or with empirical studies. It is typical to find in such studies that the running time does not vary much for M in the range from about 5 to about 25, with the running time for M in this range on

the order of 10 percent less than for the naive choice $M = 1$ (see Figure 7.8).

A slightly easier way to handle small subfiles, which is also slightly more efficient than insertion sorting them as they are encountered, is just to change the test at the beginning to

```
if (r-l <= M) return;
```

That is, we simply ignore small subfiles during partitioning. In a nonrecursive implementation, we could do so by not putting any files of size less than `M` on the stack, or, alternatively, by ignoring all files of size less than `M` that are found on the stack. After partitioning, what is left is a file that is almost sorted. As discussed in Section 6.5, however, insertion sort is the method of choice for such files. That is, insertion sort will work about as well for such a file as for the collection of little files that it would get if it were being used directly. This method should be used with caution, because insertion sort is likely to work even if quicksort has a bug that causes it not to sort at all. Excessive cost may be the only sign that something went wrong.

Figure 7.9 illustrates this process for a larger file. Even with a relatively large cutoff for small subfiles, the quicksort part of the process runs quickly because relatively few elements are involved in partitioning steps. The insertion sort that finishes the job also runs quickly because it starts with a file that is nearly in order.

This technique can be used to good advantage whenever we are dealing with a recursive algorithm. Because of their very nature, we can be sure that *all* recursive algorithms will be processing small problem instances for a high percentage of the time; we generally do have available a low-overhead brute-force algorithm for small cases; and we therefore generally can improve overall timings with a hybrid algorithm.

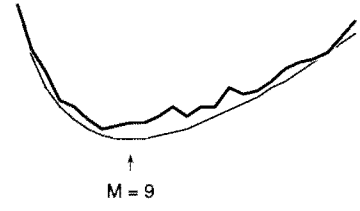


Figure 7.8
Cutoff for small subfiles

Choosing the optimal value for the cutoff for small subfiles results in about a 10 percent improvement in the average running time. Choosing the value precisely is not critical; values from a broad range (from about 5 to about 20) will work about as well for most implementations. The thick line (top) *was obtained empirically; the thin line* (bottom) *was derived analytically.*

Exercises

7.21 Are sentinel keys needed if insertion sort is called directly from within quicksort?

7.22 Instrument Program 7.1 to give the percentage of the comparisons used in partitioning files of size less than 10, 100, and 1000, and print out the percentages when you sort random files of N elements, for $N = 10^3, 10^4, 10^5$, and 10^6.

o 7.23 Implement a recursive quicksort with a cutoff to insertion sort for subfiles with less than M elements, and empirically determine the value of M

Figure 7.9
Comparisons in quicksort

Quicksort subfiles are processed independently. This picture shows the result of partitioning each subfile during a sort of 200 elements with a cutoff for files of size 15 or less. We can get a rough idea of the total number of comparisons by counting the number of marked elements by column vertically. In this case, each array position is involved in only six or seven subfiles during the sort.

for which Program 7.4 runs fastest in your computing environment to sort random files of N elements, for $N = 10^3$, 10^4, 10^5, and 10^6.

7.24 Solve Exercise 7.23 using a nonrecursive implementation.

7.25 Solve Exercise 7.23, for the case when the records to be sorted contain a key and b pointers to other information (but we are not using a pointer sort).

• 7.26 Write a program that plots a histogram (see Program 3.7) of the subfile sizes left for insertion sort when you run quicksort for a file of size N with a cutoff for subfiles of size less than M. Run your program for $M = 10$, 100, and 1000 and $N = 10^3$, 10^4, 10^5, and 10^6.

7.27 Run empirical studies to determine the average stack size used by quicksort with cutoff for files of size M, when sorting random files of N elements, for $M = 10$, 100, and 1000 and $N = 10^3$, 10^4, 10^5, and 10^6.

7.5 Median-of-Three Partitioning

Another improvement to quicksort is to use a partitioning element that is more likely to divide the file near the middle. There are several possibilities here. A safe choice to avoid the worst case is to use a random element from the array for a partitioning element. Then, the worst case will happen with negligibly small probability. This method is a simple example of a *probabilistic algorithm*—one that uses randomness to achieve good performance with high probability, regardless of the arrangement of the input. We will see numerous examples later in the book of the utility of randomness in algorithm design, particularly when bias in the input is suspected. For quicksort, it may be overkill in practice to put in a full random-number generator just for this purpose: simple arbitrary choices can also be effective.

Another well-known way to find a better partitioning element is to take a sample of three elements from the file, then to use the median of the three for the partitioning element. By choosing the three elements from the left, middle, and right of the array, we can incorporate sentinels into this scheme as well: sort the three elements (using the three-exchange method in Chapter 6), then exchange the one in the middle with `a[r-1]`, and then run the partitioning algorithm on `a[l+1]`, . . . , `a[r-2]`. This improvement is called the *median-of-three* method.

The median-of-three method helps quicksort in three ways. First, it makes the worst case much more unlikely to occur in any actual

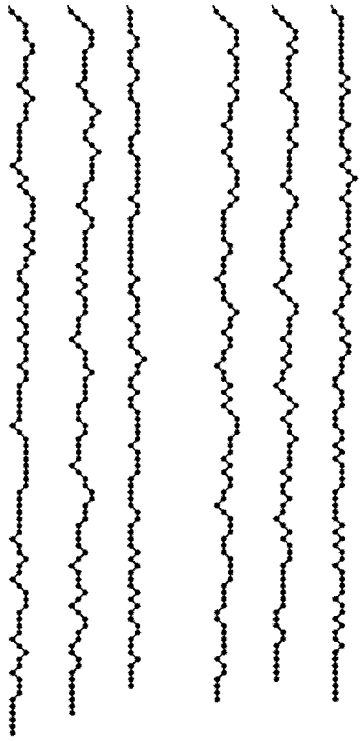


Figure 7.10
Stack size for improved versions of quicksort

Sorting the smaller subfile first guarantees that the stack size will be logarithmic at worst. Plotted here are the stack sizes for the same files as in Figure 7.5, with the smaller of the subfiles sorted first during the sort (left) *and with the median-of-three modification added* (right). *These diagrams are not indicative of running time; that variable depends on the size of the files on the stack, rather than only their number. For example, the third file (partially sorted) does not require much stack space, but leads to a slow sort because the subfiles being processed are usually large.*

sort. For the sort to take N^2 time, two out of the three elements examined must be among the largest or among the smallest elements in the file, and this event must happen consistently through most of the partitions. Second, it eliminates the need for a sentinel key for partitioning, because this function is served by one of the three elements that are examined before partitioning. Third, it reduces the total average running time of the algorithm by about 5 percent.

The combination of using the median-of-three method with a cutoff for small subfiles can improve the running time of quicksort over the naive recursive implementation by 20 to 25 percent. Program 7.4 is an implementation that incorporates all these improvements.

We might consider continuing to improve the program by removing recursion, replacing the subroutine calls by inline code, using sentinels, and so forth. However, on modern machines, such procedure calls are normally efficient, and they are not in the inner loop. More important, the use of the cutoff for small subfiles tends to compensate for any extra overhead that may be involved (outside the inner loop). The primary reason to use a nonrecursive implementation with an explicit stack is to be able to provide guarantees on limiting the stack size (see Figure 7.10).

Further algorithmic improvements are possible (for example, we could use the median of five or more elements), but the amount of time gained will be marginal for random files. We *can* realize significant time savings by coding the inner loops (or the whole program) in assembly or machine language. These observations have been validated on numerous occasions by experts with serious sorting applications (*see reference section*).

For randomly ordered files, the first exchange in Program 7.4 is superfluous. We include it not just because it leads to optimal partitioning for files already in order, but also because it protects against anomalous situations that might occur in practice (see, for example, Exercise 7.33). Figure 7.11 illustrates the effectiveness of involving the middle element in the partitioning decision, for various types of files.

The median-of-three method is a special case of the general idea that we can sample an unknown file and use properties of the sample to estimate properties of the whole file. For quicksort, we want to estimate the median to balance the partitioning. It is the nature of the algorithm that we do not need a particularly good estimate (and may

Program 7.4 Improved quicksort

Choosing the median of the first, middle, and final elements as the partitioning element and cutting off the recursion for small subfiles can significantly improve the performance of quicksort. This implementation partitions on the median of the first, middle, and final elements in the array (otherwise leaving these elements out of the partitioning process). Files of size 11 or smaller are ignored during partitioning; then, `insertion` from Chapter 6 is used to finish the sort.

```
#define M 10
void quicksort(Item a[], int l, int r)
  { int i;
    if (r-l <= M) return;
    exch(a[(l+r)/2], a[r-1]);
    compexch(a[l], a[r-1]);
      compexch(a[l], a[r]);
        compexch(a[r-1], a[r]);
    i = partition(a, l+1, r-1);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
void sort(Item a[], int l, int r)
  {
    quicksort(a, l, r);
    insertion(a, l, r);
  }
```

not want one if such an estimate is expensive to compute); we just want to avoid a particularly bad estimate. If we use a random sample of just one element, we get a randomized algorithm that is virtually certain to run quickly, no matter what the input. If we randomly choose three or five elements from the file, then use the median of that sample for partitioning, we get a better partition, but the improvement is offset by the cost of taking the sample.

Quicksort is widely used because it runs well in a variety of situations. Other methods might be more appropriate for particular cases that might arise, but quicksort handles more types of sorting problems than are handled by many other methods, and it is often significantly

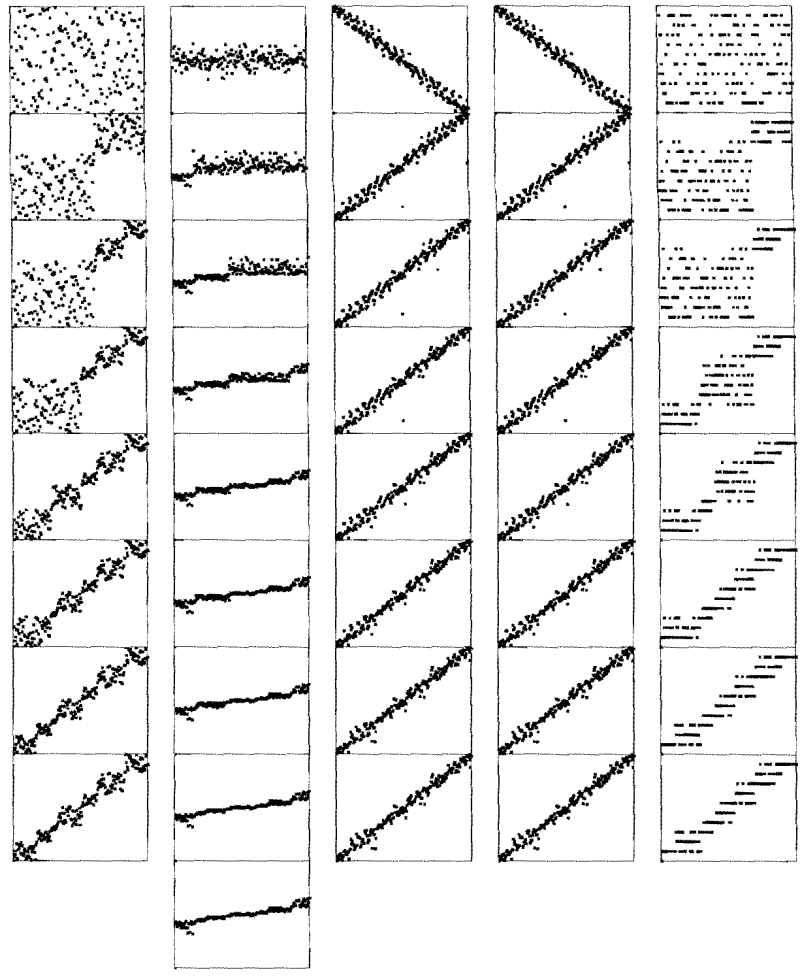


Figure 7.11
Dynamic characteristics of median-of-three quicksort on various types of files

The median-of three modification (particularly, using the middle element of the file) does a good job of making the partitioning process more robust. The degenerate types of files shown in Figure 7.4 are handled particularly well. Another option that achieves this same goal is to use a random partitioning element.

faster than alternative approaches. Table 7.1 gives empirical results in support of some of these comments.

Exercises

7.28 Our implementation of the median-of-three method is careful to ensure that the sampled elements do not participate in the partitioning process. One reason is that they can serve as sentinels. Give another reason.

7.29 Implement a quicksort based on partitioning on the median of a random sample of five elements from the file. Make sure that the elements of the sample do not participate in partitioning (see Exercise 7.28). Compare the

Table 7.1 Empirical study of basic quicksort algorithms

Quicksort (Program 7.1) is more than twice as fast as shellsort (Program 6.6) for large randomly ordered files. A cutoff for small subfiles and the median-of-three improvement (Program 7.4) lower the running time by about 10 percent each.

		Basic quicksort			Median-of-three quicksort		
N	shellsort	$M = 0$	$M = 10$	$M = 20$	$M = 0$	$M = 10$	$M = 20$
12500	6	2	2	2	3	2	3
25000	10	5	5	5	5	4	6
50000	26	11	10	10	12	9	14
100000	58	24	22	22	25	20	28
200000	126	53	48	50	52	44	54
400000	278	116	105	110	114	97	118
800000	616	255	231	241	252	213	258

performance of your algorithm with the median-of-three method for large random files.

7.30 Run your program from Exercise 7.29 on large nonrandom files—for example, sorted files, files in reverse order, or files with all keys equal. How does its performance for these files differ from its performance for random files?

•• 7.31 Implement a quicksort based on using a sample of size $2^k - 1$. First, sort the sample, then, arrange to have the recursive routine partition on the median of the sample and to move the two halves of the rest of the sample to each subfile, such that they can be used in the subfiles, without having to be sorted again. This algorithm, which uses about $N \lg N$ comparisions when k is about $\lg N - \lg \lg N$, is called *samplesort*.

•• 7.32 Run empirical studies to determine the best value of the sample size in samplesort (see Exercise 7.31), for $N = 10^3$, 10^4, 10^5, and 10^6. Does it matter whether quicksort or samplesort is used to sort the sample?

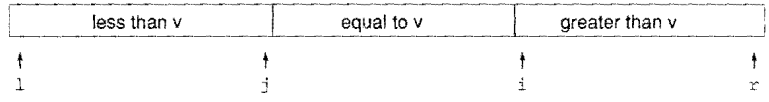
• 7.33 Show that Program 7.4, if changed to omit the first exchange and to scan over keys equal to the partitioning element, runs in quadratic time on a file that is in reverse order.

7.6 Duplicate Keys

Files with large numbers of duplicate sort keys arise frequently in applications. For example, we might wish to sort a large personnel file by year of birth, or even to use a sort to separate females from males.

When there are many duplicate keys present in the file to be sorted, the quicksort implementations that we have considered do not have unacceptably poor performance, but they can be substantially improved. For example, a file that consists solely of keys that are equal (just one value) does not need to be processed further, but our implementations so far keep partitioning down to small subfiles, no matter how big the file is (see Exercise 7.8). In a situation where there are large numbers of duplicate keys in the input file, the recursive nature of quicksort ensures that subfiles consisting solely of items with a single key value will occur often, so there is potential for significant improvement.

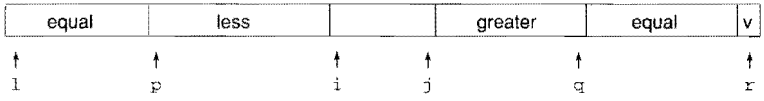
One straightforward idea is to partition the file into *three* parts, one each for keys smaller than, equal to, and larger than the partitioning element:



Accomplishing this partitioning is more complicated than the two-way partitioning that we have been using, and various different methods have been suggested for the task. It was a classical programming exercise popularized by Dijkstra as the *Dutch National Flag problem*, because the three possible key categories might correspond to the three colors on the flag (*see reference section*). For quicksort, we add the constraint that a single pass through the file must do the job—an algorithm that involves two passes through the data would slow down quicksort by a factor of two, even if there are no duplicate keys at all.

A clever method invented by Bentley and McIlroy in 1993 for three-way partitioning works by modifying the standard partitioning scheme as follows: Keep keys equal to the partitioning element that are encountered in the left subfile at the left end of the file, and keep keys equal to the partitioning element that are encountered in the right

subfile at the right end of the file. During the partitioning process, we maintain the following situation:



Then, when the pointers cross and the precise location for the equal keys is known, we swap into position all the items with keys equal to the partitioning element. This scheme does not quite meet the requirement that three-way partitioning be accomplished in one pass through the file, but the extra overhead for duplicate keys is proportional to only the number of duplicate keys found. This fact has two implications: First, the method works well even if there are no duplicate keys, since there is no extra overhead. Second, the method is linear time when there is only a constant number of key values: Each partitioning phase removes from the sort all the keys with the same value as the partitioning element, so each key can be involved in at most a constant number of partitions.

Figure 7.12 illustrates the three-way partitioning algorithm on a sample file, and Program 7.5 is a quicksort implementation based on the method. The implementation requires the addition of just two `if` statements in the exchange loop, and just two `for` loops to complete partitioning by putting the keys equal to the partitioning element into position. It seems to require less code than other alternatives for maintaining three partitions. More important, it not only handles duplicate keys in as efficient a manner as possible, but also incurs a minimal amount of extra overhead in the case that there are no duplicate keys.

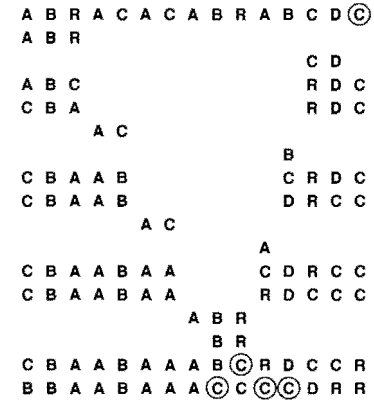


Figure 7.12
Three-way partitioning

This diagram depicts the process of putting all keys equal to the partitioning element into position. As in Figure 7.2, we scan from the left to find an element that is not smaller than the partitioning element and from the right to find an element that is not larger than the partitioning element, then exchange them. If the element on the left after the exchange is equal to the partitioning element, we exchange it to the left end of the array; we proceed similarly on the right. When the pointers cross, we put the partitioning element into position as before (next-to-bottom line)*, then exchange all the keys equal to it into position on either side of it* (bottom line).

Exercises

▷7.34 Explain what happens when Program 7.5 is run on a randomly ordered file with (*i*) two distinct key values, and (*ii*) three distinct key values.

7.35 Modify Program 7.1 to `return` if all keys in the subfile are equal. Compare the performance of your program to Program 7.1 for large random files with keys having t distinct values for $t = 2$, 5, and 10.

7.36 Suppose that we scan over keys equal to the partitioning element in Program 7.2 instead of stopping the scans when we encounter them. Show that the running time of Program 7.1 would be quadratic in this case.

Program 7.5 Quicksort with three-way partitioning

This program is based on partitioning the array into three parts: elements smaller than the partitioning element (in `a[l], ..., a[j]`); elements equal to the partitioning element (in `a[j+1], ..., a[i-1]`); and elements larger than the partitioning element (in `a[i], ..., a[r]`). Then the sort can be completed with two recursive calls, one for the smaller keys and one for the larger keys.

To accomplish the objective, the program keeps keys equal to the partitioning element on the left between `l` and `ll` and on the right between `rr` and `r`. In the partitioning loop, after the scan pointers stop and the items at `i` and `j` are exchanged, it checks each of those items to see whether it is equal to the partitioning element. If the one now on the left is equal to the partitioning element, it is exchanged into the left part of the array; if one now on the right is equal to the partitioning element, it is exchanged into the right part of the array.

After the pointers cross, the two ends of the array with elements equal to the partitioning element are exchanged back to the middle. Then those keys are in position and can be excluded from the subfiles for the recursive calls.

```
#define eq(A, B) (!less(A, B) && !less(B, A))
void quicksort(Item a[], int l, int r)
  { int i, j, k, p, q; Item v;
    if (r <= l) return;
    v = a[r]; i = l-1; j = r; p = l-1; q = r;
    for (;;)
      {
        while (less(a[++i], v)) ;
        while (less(v, a[--j])) if (j == l) break;
        if (i >= j) break;
        exch(a[i], a[j]);
        if (eq(a[i], v)) { p++; exch(a[p], a[i]); }
        if (eq(v, a[j])) { q--; exch(a[q], a[j]); }
      }
    exch(a[i], a[r]); j = i-1; i = i+1;
    for (k = l  ; k < p; k++, j--) exch(a[k], a[j]);
    for (k = r-1; k > q; k--, i++) exch(a[k], a[i]);
    quicksort(a, l, j);
    quicksort(a, i, r);
  }
```

• 7.37 Prove that the running time of the program in Exercise 7.36 is quadratic for all files with $O(1)$ distinct key values.

7.38 Write a program to determine the number of distinct keys that occur in a file. Use your program to count the distinct keys in random files of N integers in the range 0 to $M-1$, for $M = 10$, 100, and 1000, and for $N = 10^3$, 10^4, 10^5, and 10^6.

7.7 Strings and Vectors

When the sort keys are strings, we could use an abstract-string type implementation like Program 6.11 with the quicksort implementations in this chapter. Although this approach provides a correct and efficient implementation (faster than any other method we have seen so far, for large files), there is a hidden cost that is interesting to consider.

The problem lies in the cost of the `strcmp` function, which always compares two strings by proceeding from left to right, comparing strings character by character, taking time proportional to the number of leading characters that match in the two strings. For the later partitioning stages of quicksort, when keys are close together, this match might be relatively long. As usual, because of the recursive nature of quicksort, nearly all the cost of the algorithm is incurred in the later stages, so examining improvements there is worthwhile.

For example, consider a subfile of size 5 containing the keys `discreet`, `discredit`, `discrete`, `discrepancy`, and `discretion`. All the comparisons used for sorting these keys examine at least seven characters, when it would suffice to start at the seventh character, if the extra information that the first six characters are equal were available.

The three-way partitioning procedure that we considered in Section 7.6 provides an elegant way to take advantage of this observation. At each partitioning stage, we examine just one character (say the one at position `d`), assuming that the keys to be sorted are equal in positions `0` through `d-1`. We do a three-way partition with keys whose `d`th character is smaller than the `d`th character of the partitioning element on the left, those whose `d`th character is equal to the `d`th character of the partitioning element in the middle, and those whose `d`th character is larger than the `d`th character of the partitioning element on the right. Then, we proceed as usual, *except* that we sort the middle sub-

Table 7.2 Empirical study of quicksort variants

This table gives relative costs for several different versions of quicksort on the task of sorting the first N words of *Moby Dick*. Using insertion sort directly for small subfiles, or ignoring them and insertion sorting the same file afterward, are equally effective strategies, but the cost savings is slightly less than for integer keys (see Table 7.1) because comparisons are more expensive for strings. If we do not stop on duplicate keys when partitioning, then the time to sort a file with all keys equal is quadratic; the effect of this inefficiency is noticeable on this example, because there are numerous words that appear with high frequency in the data. For the same reason, three-way partitioning is effective; it is 30 to 35 percent faster than the system sort.

N	V	I	M	Q	X	T
12500	8	7	6	10	7	6
25000	16	14	13	20	17	12
50000	37	31	31	45	41	29
100000	91	78	76	103	113	68

Key:

- V Quicksort (Program 7.1)
- I Insertion sort for small subfiles
- M Ignore small subfiles, insertion sort afterward
- Q System `qsort`
- X Scan over duplicate keys (goes quadratic when keys all equal)
- T Three-way partitioning (Program 7.5)

file, starting at character `d+1`. It is not difficult to see that this method leads to a proper sort on strings, which turns out to be very efficient (see Table 7.2). We have here a convincing example of the power of thinking (and programming) recursively.

To implement the sort, we need a more general abstract type that allows access to characters of keys. The way in which strings are handled in C makes the implementation of this method particularly straightforward. However, we defer considering the implementation in detail until Chapter 10, where we consider a variety of techniques for sorting that take advantage of the fact that sort keys can often be easily decomposed into smaller pieces.

This approach generalizes to handle multidimensional sorts, where the sort keys are vectors and the records are to be rearranged such that the first components of the keys are in order, then those with first component equal are in order by second component, and so forth. If the components do not have duplicate keys, the problem reduces to sorting on the first component; in a typical application, however, each of the components may have only a few distinct values, and three-way partitioning (moving to the next component for the middle partition) is appropriate. This case was discussed by Hoare in his original paper, and is an important application.

Exercises

7.39 Discuss the possibility of improving selection, insertion, bubble, and shell sorts for strings.

o 7.40 How many characters are examined by the standard quicksort algorithm (Program 7.1, using the string type in Program 6.11) when sorting a file consisting of N strings of length t, all of which are equal? Answer the same question for the modification proposed in the text.

7.8 Selection

An important application related to sorting but for which a full sort is not required is the operation of finding the median of a set of numbers. This operation is a common computation in statistics and in various other data-processing applications. One way to proceed would be to sort the numbers and to look at the middle one, but we can do better, using the quicksort partitioning process.

The operation of finding the median is a special case of the operation of *selection*: finding the kth smallest of a set of numbers. Because an algorithm cannot guarantee that a particular item is the kth smallest without having examined and identified the $k-1$ elements that are smaller and the $N-k$ elements that are larger, most selection algorithms can return all the k smallest elements of a file without a great deal of extra calculation.

Selection has many applications in the processing of experimental and other data. The use of the median and other *order statistics* to divide a file into smaller groups is common. Often, only a small part of a large file is to be saved for further processing; in such cases, a program that can select, say, the top 10 percent of the elements of the

Program 7.6 Selection

This procedure partitions an array about the (k-1)th smallest element (the one in a[k]): It rearranges the array to leave a[1], ..., a[k-1] less than or equal to a[k], and a[k+1], ..., a[r] greater than or equal to a[k].

For example, we could call select(a, 0, N-1, N/2) to partition the array on the median value, leaving the median in a[N/2].

```
select(Item a[], int l, int r, int k)
  { int i;
    if (r <= l) return;
    i = partition(a, l, r);
    if (i > k) select(a, l, i-1, k);
    if (i < k) select(a, i+1, r, k);
  }
```

A S O R T I N G E X A M P L E
A A E (E) T I N G O X S M P L R
L I N G O P M (R) X T S
L I G (M) O P N
A A E E L I G (M) O P N R X T S

Figure 7.13
Selection of the median

For the keys in our sorting example, partitioning-based selection uses only three recursive calls to find the median. On the first call, we seek the eighth smallest in a file of size 15, and partioning gives the fourth smallest (the E); so on the second call, we seek the fourth smallest in a file of size 11, and partitioning gives the eighth smallest (the R); so on the third call, we seek the fourth smallest in a file of size 7, and find it (the M). The file is rearranged such that the median is in place, with smaller elements to the left and larger elements to the right (equal elements could be on either side), but it is not fully sorted.

file might be more appropriate than a full sort. Another important example is the use of partitioning about the median as a first step in many divide-and-conquer algorithms.

We have already seen an algorithm that we can adapt directly to selection. If k is extremely small, then selection sort (see Chapter 6) will work well, requiring time proportional to Nk: first find the smallest element, then find the second smallest by finding the smallest of the remaining items, and so forth. For slightly larger k, we shall see methods in Chapter 9 that we could adapt to run in time proportional to $N \log k$.

A selection method that runs in linear time on the average for all values of k follows directly from the partitioning procedure used in quicksort. Recall that quicksort's partitioning method rearranges an array a[l], ..., a[r] and returns an integer i such that a[l] through a[i-1] are less than or equal to a[i], and a[i+1] through a[r] are greater than or equal to a[i]. If k is equal to i, then we are done. Otherwise, if k < i, then we need to continue working in the left subfile; if k > i, then we need to continue working in the right subfile. This approach leads immediately to the recursive program for selection that is Program 7.6. An example of this procedure in operation on a small file is given in Figure 7.13.

Program 7.7 Nonrecursive selection

A nonrecursive implementation of selection simply does a partition, then moves the left pointer in if the partition fell to the left of the position sought, or moves the right pointer in if the partition fell to the right of the position sought.

```
select(Item a[], int l, int r, int k)
  {
    while (r > l)
      { int i = partition(a, l, r);
        if (i >= k) r = i-1;
        if (i <= k) l = i+1;
      }
  }
```

Program 7.7 is a nonrecursive version that follows directly from the recursive version in Program 7.6. Since that program always ends with a single call on itself, we simply reset the parameters and go back to the beginning. That is, we remove the recursion without needing a stack, also eliminating the calculations involving k by keeping k as an array index.

Property 7.4 *Quicksort-based selection is linear time on the average.*

As we did for quicksort, we can argue (roughly) that, on an extremely large file, each partition should roughly split the array in half, so the whole process should require about $N + N/2 + N/4 + N/8 + \ldots = 2N$ comparisons. And, as it was for quicksort, this rough argument is not far from the truth. An analysis similar to, but significantly more complex than, that given in Section 7.2 for quicksort (*see reference section*) leads to the result that the average number of comparisons is about

$$2N + 2k\ln(N/k) + 2(N-k)\ln(N/(N-k)),$$

which is linear for any allowed value of k. For $k = N/2$, this formula evaluates to give the result that about $(2 + 2\ln 2)N$ comparisons are required to find the median. ■

An example showing how this method finds the median in a large file is depicted in Figure 7.14. There is only one subfile, which is cut

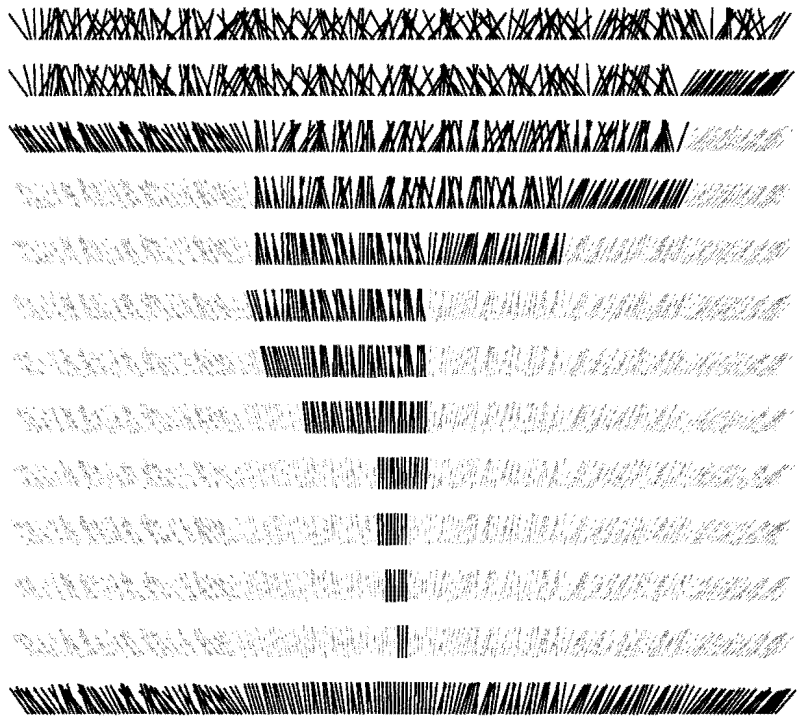


Figure 7.14
Selection of the median by partitioning

The selection process involves partitioning the subfile that contains the element sought, moving the left pointer to the right or the right pointer to the left depending on where the partition falls.

down in size by a constant factor on each call, so the procedure finishes in $O(\log N)$ steps. We can speed up the program with sampling, but we need to exercise care in doing so (see Exercise 7.45).

The worst case is about the same as for quicksort—using this method to find the smallest element in a file that is already in sorted order would result in a quadratic running time. It is possible to modify this quicksort-based selection procedure such that its running time is *guaranteed* to be linear. These modifications, although theoretically important, are extremely complex and are not at all practical.

Exercises

7.41 About how many comparisons are required, on the average, to find the smallest of N elements using `select`?

7.42 About how many comparisons are required, on the average, to find the αNth smallest element using `select`, for $\alpha = 0.1, 0.2, \ldots, 0.9$?

7.43 How many comparisions are required in the worst case to find the median of N elements using `select`?

7.44 Write an efficient program to rearrange a file such that all the elements with keys equal to the median are in place, with smaller elements to the left and larger elements to the right.

•• **7.45** Investigate the idea of using sampling to improve selection. *Hint*: Using the median may not always be helpful.

• **7.46** Implement a selection algorithm based on three-way partitioning for large random files with keys having t distinct values for $t = 2$, 5, and 10.