the transpose of 2, 4, 1, 5, 3 is 3, 5, 1, 4, 2. Each permutation has a unique transpose and is distinct from its transpose (for $n > 1$). Let i and j be integers between 1 and n, and suppose $j < i$. Then (i, j) is an inversion in exactly one of the permutations π and transpose of π. There are $n(n-1)/2$ such pairs of integers. Hence each pair of permutations has $n(n-1)/2$ inversions between them, and therefore an average of $n(n-1)/4$. Thus, overall, the average number of inversions in a permutation is $n(n-1)/4$, and we have proved the following theorem.

Theorem 4.1 Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must do at least $n(n-1)/2$ comparisons in the worst case and at least $n(n-1)/4$ comparisons on the average (for n elements). □

Since Insertion Sort does $n(n-1)/2$ key comparisons in the worst case and approximately $n^2/4$ on the average, it is about the best we can do with any algorithm that works "locally," for example, interchanging only adjacent elements. It is, of course, not obvious at this point that any other strategy can do better, but if there are significantly faster algorithms they must move elements more than one position at a time.

4.3 Divide and Conquer

The principle behind the Divide-and-Conquer algorithm design paradigm is that it is (often) easier to solve several small instances of a problem than one large one. Algorithms in Sections 4.4 through 4.8 use the Divide-and-Conquer approach. They *divide* the problem into smaller instances of the same problem (in this case into smaller sets to be sorted), then solve (*conquer*) the smaller instances recursively (i.e., by the same method), and finally *combine* the solutions to obtain the solution for the original input. To escape from the recursion, we solve some small instances of the problem directly. In contrast, Insertion Sort just "chipped off" one element and created one subproblem.

We have already seen one prime example of Divide and Conquer—Binary Search (Section 1.6). The main problem was divided into two subproblems, one of which did not even have to be solved.

In general, we can describe Divide and Conquer by the skeleton procedure in Figure 4.6.

To design a specific Divide-and-Conquer algorithm, we must specify the subroutines directlySolve, divide, and combine. The number of smaller instances into which the input is divided is k. For an input of size n, let $B(n)$ be the number of steps done by directlySolve, let $D(n)$ be the number of steps done by divide, and let $C(n)$ be the number of steps done by combine. Then the general form of the recurrence equation that describes the amount of work done by the algorithm is

$$T(n) = D(n) + \sum_{i=1}^{k} T(size(I_i)) + C(n) \qquad \text{for } n > \text{smallSize}$$

```
solve($I$)
    $n$ = size($I$);
    if ($n \leq$ smallSize)
        solution = directlySolve($I$);
    else
        divide $I$ into $I_1, \ldots, I_k$.
        for each $i \in \{1, \ldots, k\}$:
            $S_i$ = solve($I_i$);
        solution = combine($S_1, \ldots, S_k$);
    return solution;
```

Figure 4.6 The Divide-and-Conquer skeleton.

with base cases $T(n) = B(n)$ for $n \leq$ smallSize. For many Divide-and-Conquer algorithms, either the divide step or the combine step is very simple, and the recurrence equation for T is simpler than the general form. The Master Theorem (Theorem 3.17) gives solutions for a wide range of Divide-and-Conquer recurrence equations.

Quicksort and Mergesort, the sorting algorithms presented in the next few sections, differ in the ways they divide the problem and later combine the solutions, or sorted subsets. Quicksort is characterized as "hard division, easy combination," while Mergesort is characterized as "easy division, hard combination." Aside from the bookkeeping of procedure calls, we will see that all the "real work" is done in the "hard" section. Both sorting procedures have subroutines to do their "hard" section, and these subroutines are useful in their own rights. For Quicksort, the workhorse is partition, and it is the divide step in the general framework; the combine step does nothing. For Mergesort, the workhorse is merge, and it is the combine step; the divide step does one simple calculation. Both algorithms divide the problem into two subproblems. However, with Mergesort, those two subproblems are of equal size (within a margin of one element), whereas with Quicksort, an even subdivision is not assured. This difference leads to markedly different performance characteristics, which will be discovered during analysis of the respective algorithms.

At the top level, HeapSort (Section 4.8) is not a Divide-and-Conquer algorithm, but uses heap operations that are in the Divide-and-Conquer category. The accelerated form of Heapsort uses a more sophisticated Divide-and-Conquer algorithm.

In later chapters, the Divide-and-Conquer strategy will come up in numerous problems. In Chapter 5, it is applied to the problem of finding the median element of a set. (The general problem is called the selection problem.) In Chapter 6, we will use Divide and Conquer in the form of binary search trees, and their balanced versions, red-black trees. In Chapter 9, we will apply it to problems of paths in graphs, such as transitive closure. In Chapter 12, we will use it on several matrix and vector problems. In Chapter 13, we will apply it to approximate graph coloring. In Chapter 14, it reappears in a slightly different form for parallel computation.

4.4 Quicksort

Quicksort is one of the earlier Divide-and-Conquer algorithms to be discovered; it was published by C. A. R. Hoare in 1962. It is still one of the fastest in practice.

4.4.1 The Quicksort Strategy

Quicksort's strategy is to rearrange the elements to be sorted so that all the "small" keys precede the "large" keys in the array (the "hard division" part). Then Quicksort sorts the two subranges of "small" and "large" keys recursively, with the result that the entire array is sorted. For an array implementation there is nothing to do in the "combination" step, but Quicksort can also work on lists (see Exercise 4.22), in which case the "combination" step concatenates the two lists. We describe the array implementation for simplicity.

Let E be the array of elements and let first and last be the indexes of the first and last entries, respectively, in the subrange Quicksort is currently sorting. At the top level first $= 0$ and last $= n - 1$, where n is the number of elements.

The Quicksort algorithm chooses an element, called the *pivot element*, whose key is called the pivot, from the subrange that it must sort, and "pulls it out of the way"; that is, it moves the pivot element to a local variable, leaving a *vacancy* in the array. For the moment we assume that the leftmost element in the subrange is chosen as the pivot element.

Quicksort passes the pivot (the key field only) to the Partition subroutine, which rearranges the *other* elements, finding an index splitPoint such that:

1. for first $\leq$ i < splitPoint, E[i].key < pivot;
2. and for splitPoint < i $\leq$ last, E[i].key $\geq$ pivot.

Notice that there is now a *vacancy* at splitPoint.

Then Quicksort deposits the pivot element in E[splitPoint], which is its correct position, and the pivot element is ignored in the subsequent sorting. (See Figure 4.7.) This completes the "divide" process, and Quicksort continues by calling itself recursively to solve the two subproblems created by Partition.

The Quicksort procedure may choose to partition around any key in the array between E[first] and E[last], as a preprocessing step. Whatever element is chosen is moved to a local variable named pivot, and if it is *not* E[first], then E[first] is moved into its position, ensuring that there is a vacancy at E[first] when Partition is called. Other strategies for choosing a pivot are explored in Section 4.4.4.

Algorithm 4.2 Quicksort

Input: Array E and indexes first, and last, such that elements E[i] are defined for first $\leq$ $i \leq$ last.

Output: E[first], . . ., E[last] is a sorted rearrangement of the same elements.

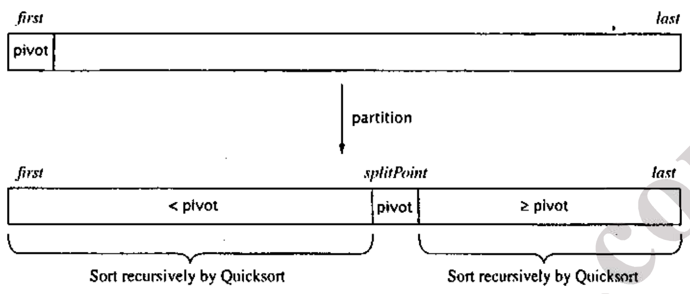

Figure 4.7 Quicksort

```
void quickSort(Element[] E, int first, int last)
    if (first < last)
        Element pivotElement = E[first];
        Key pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first, last);
        E[splitPoint] = pivotElement;
        quickSort (E, first, splitPoint - 1);
        quickSort (E, splitPoint + 1, last);
    return;
```

4.4.2 The Partition Subroutine

All the work of comparing keys and moving elements is done in the Partition subroutine. There are several different strategies that may be used by Partition; they yield algorithms with different advantages and disadvantages. We present one here and consider another in the exercises. The strategy hinges on how to carry out the rearrangement of elements. A very simple solution is to move elements into a temporary array, but the challenge is to rearrange them in place.

The partitioning method we now describe is essentially the method originally described by Hoare. As motivation, remember that the lower bound argument in Section 4.2.3 showed that, to improve on Insertion Sort, it is necessary to be able to move an element many positions after one compare. Here the vacancy is initially at E[first]. Given that we want small elements at the left end of the range, and that we want to move elements long distances whenever possible, it is very logical to start searching backward from E[last] for a small element, that is, an element less than pivot. When we find one, we move that element into the vacancy (which was at first). That leaves a new vacancy where the small element used to be; we call it highVac. The situation is illustrated in the first two array diagrams in Figure 4.8.

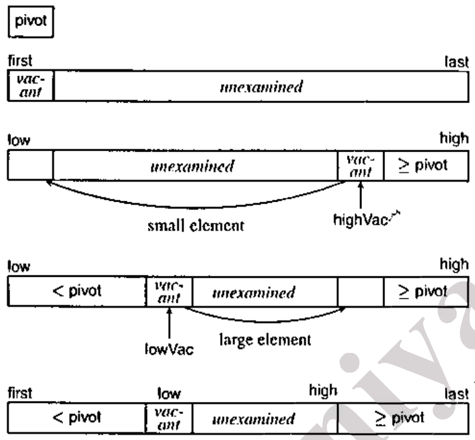

Figure 4.8 The progression of Partition through its first cycle

We know all the elements with indexes greater than highVac (through last) are greater than or equal to pivot. If possible, some other large element should be moved into highVac. Again, we want to move elements long distances, so it is logical to search forward for a *large* element this time, starting at first + 1. When we find one, we move that element into the vacancy (which was at highVac), and that leaves a new vacancy, which we call lowVac. We know all the elements with indexes less than lowVac (down to first) are less than pivot.

Finally, we update the variables low and high as indicated in the last row of Figure 4.8 to prepare for another cycle. As at the beginning of the first cycle, the elements in the range low+1 through high have not been examined yet, and E[low] is vacant. We can repeat the cycle just described, searching backward from high for a small element, moving it to the low vacancy, then searching forward from low+1 for a large element, and moving it to highVac, creating a vacancy at lowVac, the position from which the large element was moved. Eventually lowVac and highVac meet, meaning all elements have been compared with the pivot.

The Partition procedure is implemented as a repetition of the cycle just described, using subroutines to organize the code. The subroutine extendLargeRegion scans backward from the right end, passing over large elements until it either finds a small element and moves it into the vacancy at the left end, or runs into that vacancy without finding any small element. In the latter case, the partitioning is completed. In the former case, the new vacant position is returned, and the second subroutine is invoked. The subroutine extendSmallRegion is similar, except that it scans forward from the left end, passing over small elements, until it finds and moves a large element into the vacancy at the right end, or runs out of data.

Initially, the small-key region (left of low) and large-key region (right of high) are both empty, and the vacancy is at the left end of the middle region (which is the whole range at this point). Each call to a subroutine, extendLargeRegion or extendSmallRegion, shrinks the middle region by at least one, and shifts the vacancy to the other end of the middle region. The subroutines also ensure that only small elements go into the small-key region and only large elements go into the large-key region. This can be seen from their postconditions. When the middle region shrinks to one position, that position is the vacancy, and it is returned as splitPoint. It is left as an exercise to determine, line by line in the while loop of partition, what the boundaries are for the middle region and at which end the vacancy is located. Although the procedure for Partition can "make do" with fewer variables, each variable we define has its own meaning, and simplifies the answer for the exercise.

Algorithm 4.3 Partition

Input: Array E, pivot, the key around which to partition, and indexes first, and last, such that elements E[i] are defined for first $+ 1 \leq i \leq$ last and E[first] is vacant. It is assumed that first < last.

Output: Let splitPoint be the returned value. The elements originally in first+1, . . ., last are rearranged into two subranges, such that

1. the keys of E[first], . . ., E[splitPoint−1] are less than pivot, and
2. the keys of E[splitPoint+1], . . ., E[last] are greater than or equal to pivot.

Also, first $\leq$ splitPoint $\leq$ last, and E[splitPoint] is vacant.

Procedure: See Figure 4.9. ■

To avoid extra comparisons inside the while loop in partition, there is no test for highVac = lowVac before line 5, which would indicate that all elements have been partitioned. Consequently, high might be one less than low when the loop terminates, when logically it should be equal. However, high is not accessed after the loop terminates, so this difference is harmless.

A small example is shown in Figure 4.10. The detailed operation of Partition is shown only the first time it is called. Notice that the smaller elements accumulate to the left of low and the larger elements accumulate to the right of high.

4.4.3 Analysis of Quicksort

Worst case

Partition compares each key to pivot, so if there are k positions in the range of the array it is working on, it does $k - 1$ key comparisons. (The first position is vacant.) If E[first] has the smallest key in the range being split then splitPoint = first, and all that has been accomplished is splitting the range into an empty subrange (keys smaller than pivot) and

```
int partition(Element[] E, Key pivot, int first, int last)
    int low, high;
1.  low = first; high = last;
2.  while (low < high)
3.      int highVac = extendLargeRegion(E, pivot, low, high);
4.      int lowVac = extendSmallRegion(E, pivot, low+1, highVac);
5.      low = lowVac; high = highVac - 1;
6.  return low; // This is the splitPoint.

/** Postcondition for extendLargeRegion:
  * The rightmost element in E[lowVac+1], ..., E[high]
  * whose key is < pivot is moved to E[lowVac] and
  * the index from which it was moved is returned.
  * If there is no such element, lowVac is returned.
  */
int extendLargeRegion(Element[] E, Key pivot, int lowVac, int high)
    int highVac, curr;
    highVac = lowVac; // Assume failure.
    curr = high;
    while (curr > lowVac)
        if (E[curr].key < pivot)
            E[lowVac] = E[curr]; // Succeed.
            highVac = curr;
            break;
        curr --; // Keep looking.
    return highVac;

/** Postcondition for extendSmallRegion: (Exercise) */
int extendSmallRegion(Element[] E, Key pivot, int low, int highVac)
    int lowVac, curr;
    lowVac = highVac; // Assume failure.
    curr = low;
    while (curr < highVac)
        if (E[curr].key ≥ pivot)
            E[highVac] = E[curr]; // Succeed.
            lowVac = curr;
            break;
        curr ++; // Keep looking.
    return lowVac;
```

Figure 4.9 Procedure for Algorithm 4.3

```
The keys
45    14    62    51    75    96    33    84    20

"Pulling out" the pivot
45  ⟵ pivot
__    14    62    51    75    96    33    84    20

The first execution of Partition
__    14    62    51    75    96    33    84    20
↕low                                     high↕     beginning of while loop
20    14    62    51    75    96    33    84    __
                                         hVac↕     after extendLargeRegion

20    14    __    51    75    96    33    84    62
             ↖lVac                      hVac↗      after extendSmallRegion
20    14    __    51    75    96    33    84    62
            ↕low                        ↕high      beginning of while loop
20    14    33    51    75    96    __    84    62
                                    ↕hVac          after extendLargeRegion

20    14    33    __    75    96    51    84    62
                   ↖lVac             ↖hVac         after extendSmallRegion
20    14    33    __    75    96    51    84    62
                  ↕low        ↕high                beginning of while loop
20    14    33    __    75    96    51    84    62
                  ↕hVac                            after extendLargeRegion

20    14    33    __    75    96    51    84    62
              lVac↗ ↖hVac                          after extendSmallRegion
20    14    33    __    75    96    51    84    62
         high↗    ↕low                             while loop exits
20    14    33   [45]   75    96    51    84    62 Place pivot in final position

Partition first section (details not shown)
14   [20]   33

                        Partition second section (details not shown)
                        75  ⟵ pivot
                        __    96    51    84    62
                        62    51   [75]   84    96

                        Partition left subsection
                        51   [62]

                                    Partition right subsection
                                   [84]   96

Final sequence
14    20    33    45    51    62    75    84    96
```

Figure 4.10 Example of Quicksort

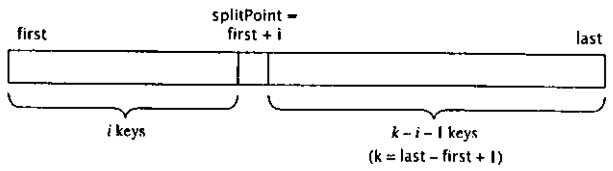

Figure 4.11 Average behavior of Quicksort.

a subrange with $k - 1$ elements. Thus, if pivot is the smallest key each time Partition is called, then the total number of key comparisons done is

$$\sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2}.$$

This is as bad as Insertion Sort and Maxsort (Exercise 4.1). And, strangely enough, the worst case occurs when the keys are already sorted in ascending order! Is the name Quicksort just a bit of false advertising?

Average Behavior

In Section 4.2.3 we showed that if a sorting algorithm removes at most one inversion from the permutation of the keys after each comparison, then it must do at least $(n^2 - n)/4$ comparisons on the average (Theorem 4.1). Quicksort, however, does not have this restriction. The Partition algorithm can move an element across a large section of the array, eliminating up to $n - 1$ inversions with one move. Quicksort deserves its name because of its average behavior.

We assume that the keys are distinct and that all permutations of the keys are equally likely. Let k be the number of elements in the range of the array being sorted, and let $A(k)$ be the average number of key comparisons done for ranges of this size. Suppose the next time Partition is executed pivot gets put in the ith position in this subrange (Figure 4.11), counting from 0. Partition does $k - 1$ key comparisons, and the subranges to be sorted next have i elements and $k - 1 - i$ elements, respectively.

It is important for our analysis that after Partition finishes, no two keys within the subrange (first, . . . , splitPoint − 1) have been compared to each other, so all permutations of keys in this subrange are still equally likely. The same holds for the subrange (splitPoint + 1, . . . , last). This justifies the following recurrence.

Each possible position for the split point i is equally likely (has probability $1/k$) so, letting $k = n$, we have the recurrence equation

$$A(n) = n - 1 + \sum_{i=0}^{n-1}\frac{1}{n}(A(i) + A(n-1-i)) \qquad \text{for } n \geq 2$$
$$A(1) = A(0) = 0.$$

Inspection of the terms in the sum lets us simplify the recurrence equation. The terms of the form $A(n-1-i)$ run from $A(n-1)$ down to $A(0)$, so their sum is the same as the sum of the $A(i)$ terms. Then we can drop the $A(0)$ terms, giving

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \qquad \text{for } n \geq 1. \tag{4.1}$$

This is a more complicated recurrence equation than the ones we saw earlier, because the value of $A(n)$ depends on all earlier values. We can try to use some ingenuity to solve the recurrence, or we can make a guess at the solution and prove it by induction. The latter technique is especially suitable for recursive algorithms. It is instructive to see both methods, so we will do both.

To form a guess for $A(n)$, let's consider a case in which Quicksort works quite well. Suppose that each time Partition is executed, it partitions the range into two equal subranges. Since we're just making an estimate to help guess how fast Quicksort is on the average, we will estimate the size of the two subranges at $n/2$ and not worry about whether this is an integer. The number of comparisons done is described by the recurrence equation

$$Q(n) \approx n + 2Q(n/2).$$

The Master Theorem (Theorem 3.17) can be applied: $b = 2$, $c = 2$, so $E = 1$, and $f(n) = n^1$. Therefore $Q(n) \in \Theta(n \log n)$. Thus if E[first] were close to the median each time the range is split, the number of comparisons done by Quicksort would be in $\Theta(n \log n)$. This is significantly better than $\Theta(n^2)$. But if all permutations of the keys are equally likely, are there enough "good" cases to affect the average? We prove that there are.

Theorem 4.2 Let $A(n)$ be defined by the recurrence equation Equation (4.1). Then, for $n \geq 1$, $A(n) \leq cn \ln n$ for some constant c. (*Note*: We have switched to the natural logarithm to simplify some of the computation in the proof. The value for c will be found in the proof.)

Proof The proof is by induction on n, the number of elements to be sorted. The base case is $n = 1$. We have $A(1) = 0$ and $c1 \ln 1 = 0$.

For $n > 1$, assume that $A(i) \leq ci \ln(i)$ for $1 \leq i < n$, for the same constant c stated in the theorem. By Equation (4.1) and the induction hypothesis,

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \leq n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} ci \ln(i).$$

We can bound the sum by integrating (see Equation 1.16):

$$\sum_{i=1}^{n-1} ci \ln(i) \leq c\int_1^n x \ln x \, dx.$$

Using Equation (1.15) from Section 1.3.2 gives

$$\int_1^n x \ln x \, dx = \tfrac{1}{2}n^2 \ln(n) - \frac{1}{4}n^2$$

so

$$A(n) \le n - 1 + \frac{2c}{n}\left(\tfrac{1}{2}n^2 \ln(n) - \frac{1}{4}n^2\right)$$
$$= cn \ln n + n(1 - \frac{c}{2}) - 1.$$

To show that $A(n) \le cn \ln n$, it suffices to show that the second and third terms are negative or zero. The second term is less than or equal to zero for $c \ge 2$. So we can let $c = 2$ and conclude that $A(n) \le 2\,n \ln n$. □

A similar analysis shows that $A(n) > cn \ln n$ for any $c < 2$. Since $\ln n \approx 0.693 \lg n$, we therefore have:

Corollary 4.3 On average, assuming all input permutations are equally likely, the number of comparisons done by Quicksort (Algorithm 4.2) on sets of size n is approximately $1.386\,n \lg n$, for large n. □

* Average Behavior, More Exactly

Although we have established the average behavior of Quicksort, it is still instructive to return to the recurrence equation (Equation 4.1) and try to solve it directly, getting more than the leading term. This section uses some sophisticated mathematics, and can be omitted without loss of continuity.

We have, by Equation (4.1),

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i). \qquad (4.2)$$

$$A(n-1) = n - 2 + \frac{2}{n-1}\sum_{i=1}^{n-2} A(i). \qquad (4.3)$$

If we subtract the summation in Equation (4.3) from the summation in Equation (4.2), most of the terms drop out. Since the summations are multiplied by different factors, we need a slightly more complicated bit of algebra. Informally, we compute

$$n \times \text{Equation (4.2)} - (n-1) \times \text{Equation (4.3)}.$$

So

$$nA(n) - (n-1)A(n-1) = n(n-1) + 2\sum_{i=1}^{n-1} A(i) - (n-1)(n-2) - 2\sum_{i=1}^{n-2} A(i)$$
$$= 2A(n-1) + 2(n-1).$$

So

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

Now let

$$B(n) = \frac{A(n)}{n+1}.$$

The recurrence equation for B is

$$B(n) = B(n-1) + \frac{2(n-1)}{n(n+1)} \qquad B(1) = 0.$$

With the aid of Equation (1.11), we leave it for readers to verify that

$$B(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2\sum_{i=1}^{n} \frac{1}{i} - 4\sum_{i=1}^{n} \frac{1}{i(i+1)}$$
$$\approx 2(\ln n + 0.577) - 4n/(n+1).$$

Therefore

$$A(n) \approx 1.386\, n \lg n - 2.846\, n.$$

Space usage

At first glance it may seem that Quicksort is an in-place sort. It is not. While the algorithm is working on one subrange, the beginning and ending indexes (call them the borders) of all the other subranges yet to be sorted are saved on the frame stack, and the size of the stack depends on the number of subranges into which the range will be split. This, of course, depends on n. In the worst case, Partition splits off one entry at a time; the depth of the recursion is n. Thus the worst-case amount of space used by the stack is in $\Theta(n)$. One of the modifications to the algorithm described next can significantly reduce the maximum stack size.

4.4.4 Improvements on the Basic Quicksort Algorithm

Choice of Pivot

We have seen that Quicksort works well if the pivot key used by Partition to partition a segment belongs near the middle of the segment. (Its position is the value, splitPoint, returned by Partition.) Choosing E[first] as the pivot element causes Quicksort to do poorly in cases where sorting should be easy (for example, when the array is already sorted). There are several other strategies for choosing the pivot element. One is to choose a random integer q between first and last and let pivot = E[q].key. Another is to let pivot be the median key of the entries E[first], E[(first+last)/2], and E[last]. (In either case, the element in E[first] would be swapped with the pivot element before proceeding with the Partition algorithm.) Both of these strategies require some extra work to choose pivot, but they pay off by improving the average running time of a Quicksort program.

Alternative Partition Strategy

The version of Partition presented in the text does the fewest element movements, on average, compared to other partitioning strategies. It is shown with subroutines for clarity, and coding these in-line would save some overhead; however, some optimizing compilers can make this change automatically. Other optimization considerations are mentioned in Notes and References at the end of the chapter. There is an alternative version in the exercises that is easy to understand and program, but somewhat slower.

Small Sort

Quicksort is not particularly good for small sets, due to the overhead of procedure calls. But, by the nature of the algorithm, for large *n* Quicksort will break the set up into small subsets and recursively sort them. Thus whenever the size of a subset is small, the algorithm becomes inefficient. This problem can be remedied by choosing a small smallSize and sorting subsets of size less than or equal to smallSize by some simple, nonrecursive sort, called smallSort in the modified algorithm. (Insertion Sort is a good choice.)

```
quickSort(E, first, last)
    if (last − first > smallSize)
        pivotElement = E[first];
        pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first, last);
        E[splitPoint] = pivotElement;
        quickSort(E, first, splitPoint − 1);
        quickSort(E, splitPoint + 1, last);
    else
        smallSort(E, first, last);
```

A variation on this theme is to skip calling smallSort. Then when Quicksort exits, the array is not sorted, but no element needs to move more than smallSize places to reach its correct sorted position. (Why not?) Therefore one postprocessing run of Insertion Sort will be very efficient, and will do about the same comparisons as all the calls to it in its role as smallSort.

What value should smallSize have? The best choice depends on the particular implementation of the algorithm (that is, the computer being used and the details of the program), since we are making some trade-offs between overhead and key comparisons. A value close to 10 may do reasonably well.

Stack Space Optimization

We observed that the depth of recursion for Quicksort can grow quite large, proportional to *n* in the worst case (when Partition splits off only one element each time). Much of the pushing and popping of the frame stack that will be done is unnecessary. After Partition, the program starts sorting the subrange E[first], . . ., E[splitPoint − 1]; later it must sort the subrange E[splitPoint+1], . . ., E[last].

The second recursive call is the last statement in the procedure, so it can be converted into iteration in the manner we have seen earlier for shiftVac in Insertion Sort. The first recursive call remains, so the recursion is only partially eliminated.

With only one recursive call left in the procedure, we still need to be concerned about excessive depth of recursion. This can occur through a succession of recursive calls that each work on a subrange only slightly smaller than the preceding one. So the second trick we use is to avoid making the recursive call on the *larger* subrange. By ensuring that each recursive call is on at most half as many elements as its "parent" call, the depth of recursion is guaranteed to remain within about lg n. The two ideas are combined in the following version, in which "TRO" stands for "tail recursion optimization." The idea is that after each partition, the next recursive call will work on the smaller subrange, and the larger subrange will be handled directly in the while loop.

```
quickSortTRO(E, first, last)
    int first1, last1, first2, last2;

    first2 = first; last2 = last;
    while (last2 - first2 > 1)
        pivotElement = E[first];
        pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first2, last2);
        E[splitPoint] = pivotElement;
        if (splitPoint < (first2 + last2) / 2)
            first1 = first2; last1 = splitPoint - 1;
            first2 = splitPoint + 1; last2 = last2;
        else
            first1 = splitPoint + 1; last1 = last2;
            first2 = first2; last2 = splitPoint - 1;
        quickSortTRO(E, first1, last1);
        // Continue loop for first2, last2.
    return;
```

Combined Improvements

We discussed the preceding modifications independently, but they are compatible and can be combined in one program.

Remarks

In practice, Quicksort programs run quite fast on the average for large n, and they are widely used. In the worst case, though, Quicksort behaves poorly. Like Insertion Sort (Section 4.2), Maxsort and Bubble Sort (Exercises 4.1 and 4.2), Quicksort's worst-case time is in $\Theta(n^2)$, but unlike the others, Quicksort's average behavior is in $\Theta(n \log n)$. Are there sorting algorithms whose worst-case time is in $\Theta(n \log n)$, or can we establish a worst-case lower bound of $\Theta(n^2)$? The Divide-and-Conquer approach gave us the improvement in average behavior. Let's examine the general technique again and see how to use it to improve on the worst-case behavior.

4.5 Merging Sorted Sequences

In this section we review a straightforward solution to the following problem: Given two sequences A and B sorted in nondecreasing order, merge them to create one sorted sequence C. Merging sorted subsequences is essential to the strategy of Mergesort. It also has numerous applications in its own right, some of which are covered in the exercises. The measure of work done by a merge algorithm will be the number of comparisons of keys performed by the algorithm.

Let k and m be the number of items in sequences A and B, respectively. Let $n = k + m$ be the "problem size." Assuming neither A nor B is empty, we can immediately determine the first item in C: It is the minimum between the first items of A and B. What about the rest of C? Suppose the first element of A was the minimum. Then the remainder of C must be the result of merging all elements of A *after* the first with all elements of B. But this is just a smaller version of the same problem we started with. The situation is symmetrical if the first element of B was the minimum. In either case the problem size for the remaining problem (of constructing the rest of C) is $n - 1$. Method 99 (Section 3.2.2) comes to mind.

If we assume that we only need to merge problems of size up to 100, and we can call upon merge99 to merge problems of size up to 99, then the problem is already solved. The pseudocode follows:

```
merge(A, B, C)
    if (A is empty)
        rest of C = rest of B
    else if (B is empty)
        rest of C = rest of A
    else
        if (first of A is less than first of B)
            first of C = first of A
            merge99(rest of A, B, rest of C)
        else
            first of C = first of B
            merge99(A, rest of B, rest of C)
    return
```

Now just change merge99 to merge for the general recursive solution.

Once the solution idea is seen, we can also see how to formulate an iterative solution. The idea works for all sequential data structures, but we state the algorithm in terms of arrays, for definiteness. We introduce three indexes to keep track of where "rest of A," "rest of B," and "rest of C" begin at any stage in the iteration. (These indexes would be parameters in the recursive version.)

Algorithm 4.4 Merge

Input: Arrays A with k elements and B with m elements, each in nondecreasing order of their keys.

Output: C, an array containing $n = k + m$ elements from A and B in nondecreasing order. C is passed in and the algorithm fills it.

```
void merge(Element[] A, int k, Element[] B, int m, Element[] C)
    int n = k + m;
    int indexA = 0; indexB = 0; indexC = 0;
    // indexA is the beginning of rest of A; same for B, C.

    while (indexA < k && indexB < m)
        if (A[indexA].key < B[indexB].key)
            C[indexC] = A[indexA];
            indexA ++;
            indexC ++;
        else
            C[indexC] = B[indexB];
            indexB ++;
            indexC ++;
        // Continue loop
    if (indexA ≥ k)
        Copy B[indexB, . . ., m−1] to C[indexC, . . ., n−1].
    else
        Copy A[indexA, . . ., k−1] to C[indexC, . . ., n−1].
```

4.5.1 Worst Case

Whenever a comparison of keys from A and B is done, at least one element is moved to C and never examined again. After the last comparison, at least two elements—the two just compared—have not yet been moved to C. The smaller one is moved immediately, but now C has at most $n - 1$ elements, and no more comparisons will be done. Those that remain in the other array are moved to C without any further comparisons. So at most $n - 1$ comparisons are done. The worst case, using all $n - 1$ comparisons, occurs when $A[k - 1]$ and $B[m - 1]$ belong in the last two positions in C.

4.5.2 Optimality of Merge

We show next that Algorithm 4.4 is optimal in the worst case among comparison-based algorithms when $k = m = n/2$. That is, for any comparison-based algorithm that merges correctly on all inputs for which $k = m = n/2$ there must be *some* input for which it requires $n - 1$ comparisons. (This is not to say that for a *particular* input no algorithm could do better than Algorithm 4.4.) After considering $k = m = n/2$, we look at some other relationships between k and m.

Theorem 4.4 Any algorithm to merge two sorted arrays, each containing $k = m = n/2$ entries, by comparison of keys, does at least $n - 1$ such comparisons in the worst case.

Proof Suppose we are given an arbitrary merge algorithm. Let a_i and b_i be the ith entries of A and B, respectively. We show that keys can be chosen so that the algorithm must compare a_i with b_i, for $0 \leq i < m$, and a_i with b_{i+1}, for $0 \leq i < m - 1$. Specifically, choose keys so that, whenever the algorithm compares a_i and b_j, if $i < j$, the result is that $a_i < b_j$, and if $i \geq j$, the result is that $b_j < a_i$. Choosing the keys so that

$$b_0 < a_0 < b_1 < a_1 < \cdots < b_i < a_i < b_{i+1} < \cdots < b_{m-1} < a_{m-1} \tag{4.4}$$

will satisfy these conditions. However, if for some i, the algorithm never compares a_i and b_i, then choosing keys in the same order as in Equation (4.4), except that $a_i < b_i$, will also satisfy these conditions and the algorithm would not be able to determine the correct ordering.

Similarly, if for some i, it never compares a_i and b_{i+1}, the arrangement of Equation (4.4), except that $b_{i+1} < a_i$ would be consistent with the results of the comparisons done, and again the algorithm could not determine the correct ordering. □

Can we generalize this conclusion? Suppose k and m differ slightly (as we will see they might in Mergesort)?

Corollary 4.5 Any algorithm to merge two sorted arrays by comparison of keys, where the inputs contain k and m entries, respectively, k and m differ by one, and $n = k + m$, does at least $n - 1$ such comparisons in the worst case.

Proof The same proof as in Theorem 4.4 applies, except there is no a_{m-1}. □

Can we generalize this conclusion still further? If we find one kind of behavior at one extreme, it is often a good idea to check the other extreme. Here the first "extreme" was $k = m$, so the other extreme makes k and m as different as possible. Let's look at an extreme case, where $k = 1$ and m is large, so $n = m + 1$. We can devise an algorithm that uses at most $\lceil \lg(m + 1) \rceil$ comparisons. (What is it?) So clearly $n - 1$ is not a lower bound in this case. The improvement for $k = 1$ can be generalized to other cases where k is much less than n (see Exercise 4.24). Therefore the lower bound arguments of Theorem 4.4 and Corollary 4.5 cannot be extended to all combinations of k and m. For further possibilities, look at Exercise 4.33 after reading Section 4.7.

4.5.3 Space Usage

It might appear from the way in which Algorithm 4.4 is written that merging sequences with a total of n entries requires enough memory locations for $2n$ entries, since all entries

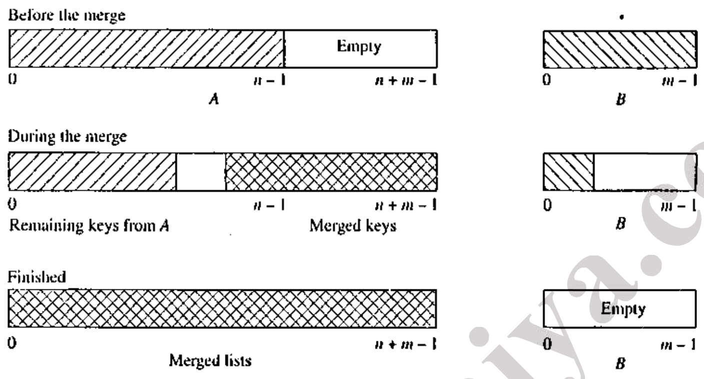

Figure 4.12 Overlapping arrays for Merge

are copied to C. In some cases, however, the amount of extra space needed can be decreased. One case is that the sequences are linked lists, and A and B are not needed (as lists) after the merge is completed. Then the list nodes of A and B can be recycled as C is created.

Suppose the input sequences are stored in arrays and suppose $k \geq m$. If A has enough room for $n = k + m$ elements, then only the extra m locations in A are needed. Simply identify C with A, and do the merging from the right ends (larger keys) of A and B, as indicated in Figure 4.12. The first m entries moved to "C" will fill the extra locations of A. From then on the vacated locations in A are used. There will always be a gap (i.e., some empty locations) between the end of the merged portion of the array and the remaining entries of A until all of the entries have been merged. Observe that if this space-saving storage layout is used, the last lines in the merge algorithm (**else** Copy A[indexA], . . ., A[k−1] to C[indexC], . . ., C[n−1]) can be eliminated because, if B empties before A, the remaining items in A are in their correct position and do not have to be moved.

Whether or not C overlaps one of the input arrays, the extra space used by the Merge algorithm when $k = m = n/2$ is in $\Theta(n)$.

4.6 Mergesort

The problem with Quicksort is that Partition doesn't always decompose the array into two equal subranges. Mergesort just slices the array in two halves and sorts the halves separately (and of course, recursively). Then it merges the sorted halves (see Figure 4.13).

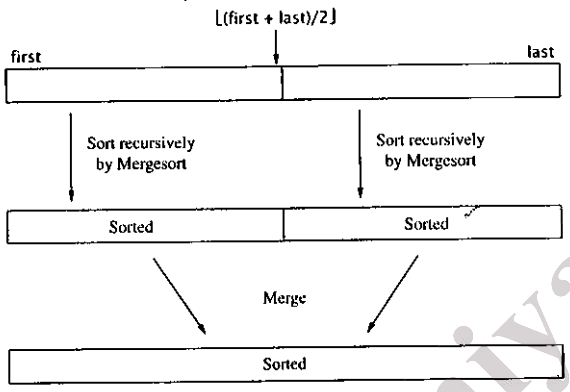

Figure 4.13 Mergesort strategy

Thus, using the divide-and-conquer terminology of Section 4.3, divide merely computes the middle index of the subrange and does no key comparisons; combine does the merging.

We assume that Merge is modified to merge adjacent subranges of one array, putting the resulting merged array back into the cells originally occupied by the elements being merged. Its parameters now are the array name E, and the first, mid, and last indexes of the subranges it is to merge; that is, the sorted subranges are E[first], . . ., E[mid] and E[mid+1], . . ., E[last], and the final sorted range is to be E[first], . . ., E[last]. In this modification, the merge subroutine is also responsible for allocating additional workspace needed. Some of the issues were discussed in Section 4.5.3.

Algorithm 4.5 Mergesort

Input: Array E and indexes first, and last, such that the elements of E[i] are defined for first $\leq i \leq$ last.

Output: E[first], . . ., E[last] is a sorted rearrangement of the same elements.

```
void mergeSort(Element[] E, int first, int last)
    if (first < last)
        int mid = (first+last) / 2;
        mergeSort(E, first, mid);
        mergeSort(E, mid + 1, last);
        merge(E, first, mid, last);
    return;
```

We have observed that students often confuse the Merge and Mergesort algorithms. Remember that Merge*sort* is a sorting algorithm. It begins with *one* scrambled array and sorts it. Merge begins with *two* arrays that are already sorted; it combines them into one sorted array.

Mergesort Analysis

First, we find the asymptotic order of the worst-case number of key comparisons for Mergesort. As usual, we define the problem size as $n = \text{last} - \text{first} + 1$, the number of elements in the range to be sorted. The recurrence equation for the worst-case behavior of Mergesort is

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \qquad (4.5)$$
$$W(1) = 0.$$

The Master Theorem tells us immediately that $W(n) \in \Theta(n \log n)$. So we finally have a sorting algorithm whose worst-case behavior is in $\Theta(n \log n)$. Rather than carry out a separate analysis of the average complexity of Mergesort, we will defer this question until we have developed the very general Theorem 4.11, concerning average behavior, in Section 4.7, just ahead.

A possible disadvantage of Mergesort is its requirement for auxiliary workspace. Because of the extra space used for the merging, which is in $\Theta(n)$, Mergesort is not an in-place sort.

★ Mergesort Analysis, More Exactly

It is of some interest to obtain a more exact estimate of the worst-case number of comparisons, in light of lower bounds to be developed in the next section (Section 4.7). We will see that Mergesort is very close to the lower bound. Readers may skip the details of this section without loss of continuity and proceed to its main conclusion, Theorem 4.6.

In the recursion tree for Equation (4.5) (see Figure 4.14), we observe that the nonrecursive costs of nodes at depth d sum to $n - 2^d$ (for all node depths not containing any base cases). We can determine that all base cases (for which $W(1) = 0$) occur at depths $\lceil \lg(n + 1) \rceil - 1$ or $\lceil \lg(n + 1) \rceil$. There are exactly n base-case nodes. Let the maximum depth be D (that is, $D = \lceil \lg(n + 1) \rceil$) and let B be the number of base cases at depth $D - 1$. Then there are $n - B$ base cases at depth D (and no other nodes at depth D). Each nonbase node at depth $D - 1$ has two children, so there are $(n - B)/2$ nonbase cases at depth $D - 1$. Using this information, we compute the sum of nonrecursive costs for the last few depths as follows:

1. Depth $D - 2$ has 2^{D-2} nodes, none of which are base cases. The sum of nonrecursive costs for this level is $n - 2^{D-2}$.
2. Depth $D - 1$ has $(n - B)/2$ nonbase cases. Each has problem size 2 (with cost 1), so the sum of the nonrecursive costs for this level is $(n - B)/2$.
3. Depth D has $n - B$ base cases, cost 0.

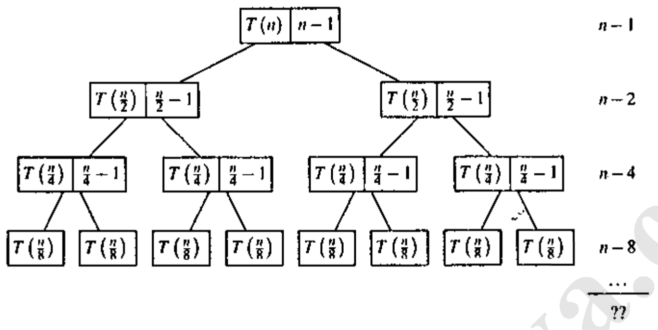
$T(n)$ | $n-1$
$T(\frac{n}{2})$ | $\frac{n}{2}-1$
$T(\frac{n}{4})$ | $\frac{n}{4}-1$
$T(\frac{n}{8})$
$n-1$
$n-2$
$n-4$
$n-8$
...
??

Figure 4.14 Recursion tree for Mergesort. Wherever a node size parameter is odd, the left child size is rounded up and the right child size is rounded down.

You can verify that $B = 2^D - n$ (Exercise 4.29). Therefore

$$\begin{aligned} W(n) &= \sum_{d=0}^{D-2} \left(n - 2^d\right) + (n - B)/2 \\ &= n(D-1) - 2^{D-1} + 1 + (n - B)/2 \\ &= n\,D - 2^D + 1. \end{aligned} \tag{4.6}$$

Because D is rounded up to an integer and occurs in the exponent, it is hard to tell how Equation (4.6) behaves between powers of 2. We prove the following theorem, which removes the ceiling function from the exponent.

Theorem 4.6 The number of comparisons done by Mergesort in the worst case is between $\lceil n \lg(n) - n + 1 \rceil$ and $\lceil n \lg(n) - .914\, n \rceil$.

Proof If we define $\alpha = 2^D/n$, then $1 \le \alpha < 2$, and D can be replaced throughout Equation (4.6) by $(\lg(n) + \lg(\alpha))$. This leads to $W(n) = n \lg(n) - (\alpha - \lg \alpha)n + 1$. The minimum value of $(\alpha - \lg \alpha)$ is about .914 (see Exercise 4.30) and the maximum in the range under consideration is 1. □

Thus Mergesort does about 30 percent fewer comparisons in the worst case than Quicksort does in the average case. However, Mergesort does more element movement than Quicksort does on average, so its time may not be faster (see Exercises 4.21 and 4.27).