Continuing in this manner, we obtain

$$T(N) = 2^k T(N/2^k) + k \cdot N$$

Using $k = \log N$, we obtain

$$T(N) = NT(1) + N \log N = N \log N + N$$

The choice of which method to use is a matter of taste. The first method tends to produce scrap work that fits better on a standard $8^1/_2 \times 11$ sheet of paper leading to fewer mathematical errors, but it requires a certain amount of experience to apply. The second method is more of a brute-force approach.

Recall that we have assumed $N = 2^k$. The analysis can be refined to handle cases when N is not a power of 2. The answer turns out to be almost identical (this is usually the case).

Although mergesort's running time is $O(N \log N)$, it has the significant problem that merging two sorted lists uses linear extra memory. The additional work involved in copying to the temporary array and back, throughout the algorithm, slows the sort considerably. This copying can be avoided by judiciously switching the roles of `a` and `tmpArray` at alternate levels of the recursion. A variant of mergesort can also be implemented nonrecursively (Exercise 7.16).

The running time of mergesort, when compared with other $O(N \log N)$ alternatives, depends heavily on the relative costs of comparing elements and moving elements in the array (and the temporary array). These costs are language dependent.

For instance, in Java, when performing a generic sort (using a `Comparator`), an element comparison can be expensive (because comparisons might not be easily inlined, and thus the overhead of dynamic dispatch could slow things down), but moving elements is cheap (because they are reference assignments, rather than copies of large objects). Mergesort uses the lowest number of comparisons of all the popular sorting algorithms, and thus is a good candidate for general-purpose sorting in Java. In fact, it is the algorithm used in the standard Java library for generic sorting.

On the other hand, in classic C++, in a generic sort, copying objects can be expensive if the objects are large, while comparing objects often is relatively cheap because of the ability of the compiler to aggressively perform inline optimization. In this scenario, it might be reasonable to have an algorithm use a few more comparisons, if we can also use significantly fewer data movements. *Quicksort,* which we discuss in the next section, achieves this tradeoff and is the sorting routine that has been commonly used in C++ libraries. New C++11 move semantics possibly change this dynamic, and so it remains to be seen whether quicksort will continue to be the sorting algorithm used in C++ libraries.

7.7 Quicksort

As its name implies for C++, **quicksort** has historically been the fastest known generic sorting algorithm in practice. Its average running time is $O(N \log N)$. It is very fast, mainly due to a very tight and highly optimized inner loop. It has $O(N^2)$ worst-case performance, but this can be made exponentially unlikely with a little effort. By combining quicksort

with heapsort, we can achieve quicksort's fast running time on almost all inputs, with heapsort's $O(N \log N)$ worst-case running time. Exercise 7.27 describes this approach.

The quicksort algorithm is simple to understand and prove correct, although for many years it had the reputation of being an algorithm that could in theory be highly optimized but in practice was impossible to code correctly. Like mergesort, quicksort is a divide-and-conquer recursive algorithm.

Let us begin with the following simple sorting algorithm to sort a list. Arbitrarily choose any item, and then form three groups: those smaller than the chosen item, those equal to the chosen item, and those larger than the chosen item. Recursively sort the first and third groups, and then concatenate the three groups. The result is guaranteed by the basic principles of recursion to be a sorted arrangement of the original list. A direct implementation of this algorithm is shown in Figure 7.13, and its performance is, generally speaking, quite

```
1   template <typename Comparable>
2   void SORT( vector<Comparable> & items )
3   {
4       if( items.size( ) > 1 )
5       {
6           vector<Comparable> smaller;
7           vector<Comparable> same;
8           vector<Comparable> larger;
9
10          auto chosenItem = items[ items.size( ) / 2 ];
11
12          for( auto & i : items )
13          {
14              if( i < chosenItem )
15                  smaller.push_back( std::move( i ) );
16              else if( chosenItem < i )
17                  larger.push_back( std::move( i ) );
18              else
19                  same.push_back( std::move( i ) );
20          }
21
22          SORT( smaller );     // Recursive call!
23          SORT( larger );      // Recursive call!
24
25          std::move( begin( smaller ), end( smaller ), begin( items ) );
26          std::move( begin( same ), end( same ), begin( items ) + smaller.size( ) );
27          std::move( begin( larger ), end( larger ), end( items ) - larger.size( ) );
28      }
29  }
```

Figure 7.13 Simple recursive sorting algorithm

respectable on most inputs. In fact, if the list contains large numbers of duplicates with relatively few distinct items, as is sometimes the case, then the performance is extremely good.

The algorithm we have described forms the basis of the quicksort. However, by making the extra lists, and doing so recursively, it is hard to see how we have improved upon mergesort. In fact, so far, we really haven't. In order to do better, we must avoid using significant extra memory and have inner loops that are clean. Thus quicksort is commonly written in a manner that avoids creating the second group (the equal items), and the algorithm has numerous subtle details that affect the performance; therein lies the complications.

We now describe the most common implementation of quicksort—"classic quicksort," in which the input is an array, and in which no extra arrays are created by the algorithm.

The classic quicksort algorithm to sort an array S consists of the following four easy steps:

1. If the number of elements in S is 0 or 1, then return.
2. Pick any element v in S. This is called the **pivot.**
3. **Partition** $S - \{v\}$ (the remaining elements in S) into two disjoint groups: $S_1 = \{x \in S - \{v\} | x \leq v\}$, and $S_2 = \{x \in S - \{v\} | x \geq v\}$.
4. Return {quicksort(S_1) followed by v followed by quicksort(S_2)}.

Since the partition step ambiguously describes what to do with elements equal to the pivot, this becomes a design decision. Part of a good implementation is handling this case as efficiently as possible. Intuitively, we would hope that about half the elements that are equal to the pivot go into S_1 and the other half into S_2, much as we like binary search trees to be balanced.

Figure 7.14 shows the action of quicksort on a set of numbers. The pivot is chosen (by chance) to be 65. The remaining elements in the set are partitioned into two smaller sets. Recursively sorting the set of smaller numbers yields $0, 13, 26, 31, 43, 57$ (by rule 3 of recursion). The set of large numbers is similarly sorted. The sorted arrangement of the entire set is then trivially obtained.

It should be clear that this algorithm works, but it is not clear why it is any faster than mergesort. Like mergesort, it recursively solves two subproblems and requires linear additional work (step 3), but, unlike mergesort, the subproblems are not guaranteed to be of equal size, which is potentially bad. The reason that quicksort is faster is that the partitioning step can actually be performed in place and very efficiently. This efficiency more than makes up for the lack of equal-sized recursive calls.

The algorithm as described so far lacks quite a few details, which we now fill in. There are many ways to implement steps 2 and 3; the method presented here is the result of extensive analysis and empirical study and represents a very efficient way to implement quicksort. Even the slightest deviations from this method can cause surprisingly bad results.

7.7.1 Picking the Pivot

Although the algorithm as described works no matter which element is chosen as pivot, some choices are obviously better than others.

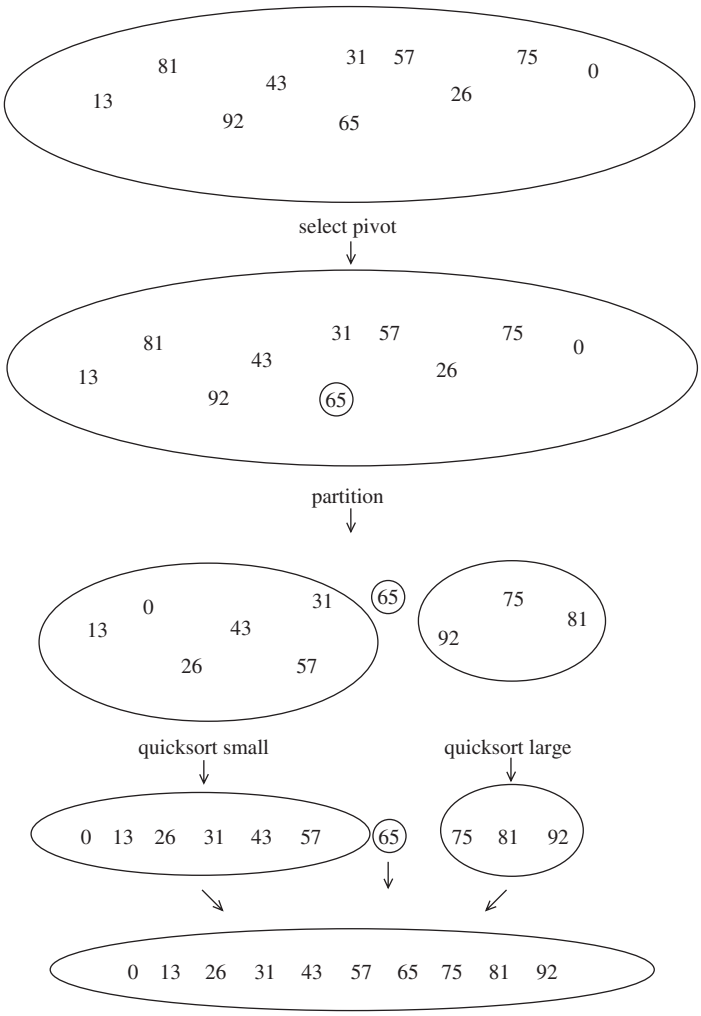

Figure 7.14 The steps of quicksort illustrated by example

A Wrong Way

The popular, uninformed choice is to use the first element as the pivot. This is acceptable if the input is random, but if the input is presorted or in reverse order, then the pivot provides a poor partition, because either all the elements go into S_1 or they go into S_2. Worse, this happens consistently throughout the recursive calls. The practical effect is that if the first element is used as the pivot and the input is presorted, then quicksort will take quadratic time to do essentially nothing at all, which is quite embarrassing. Moreover, presorted input (or input with a large presorted section) is quite frequent, so using the first element as pivot *is an absolutely horrible idea* and should be discarded immediately. An alternative is choosing the larger of the first two distinct elements as pivot, but this has

the same bad properties as merely choosing the first element. Do not use that pivoting strategy, either.

A Safe Maneuver

A safe course is merely to choose the pivot randomly. This strategy is generally perfectly safe, unless the random number generator has a flaw (which is not as uncommon as you might think), since it is very unlikely that a random pivot would consistently provide a poor partition. On the other hand, random number generation is generally an expensive commodity and does not reduce the average running time of the rest of the algorithm at all.

Median-of-Three Partitioning

The median of a group of N numbers is the $\lceil N/2 \rceil$th largest number. The best choice of pivot would be the median of the array. Unfortunately, this is hard to calculate and would slow down quicksort considerably. A good estimate can be obtained by picking three elements randomly and using the median of these three as pivot. The randomness turns out not to help much, so the common course is to use as pivot the median of the left, right, and center elements. For instance, with input $8, 1, 4, 9, 6, 3, 5, 2, 7, 0$ as before, the left element is 8, the right element is 0, and the center (in position $\lfloor (left + right)/2 \rfloor$) element is 6. Thus, the pivot would be $v = 6$. Using median-of-three partitioning clearly eliminates the bad case for sorted input (the partitions become equal in this case) and actually reduces the number of comparisons by 14%.

7.7.2 Partitioning Strategy

There are several partitioning strategies used in practice, but the one described here is known to give good results. It is very easy, as we shall see, to do this wrong or inefficiently, but it is safe to use a known method. The first step is to get the pivot element out of the way by swapping it with the last element. `i` starts at the first element and `j` starts at the next-to-last element. If the original input was the same as before, the following figure shows the current situation:

8	1	4	9	0	3	5	2	7	6
↑								↑	
i								j	

For now, we will assume that all the elements are distinct. Later on, we will worry about what to do in the presence of duplicates. As a limiting case, our algorithm must do the proper thing if *all* of the elements are identical. It is surprising how easy it is to do the *wrong* thing.

What our partitioning stage wants to do is to move all the small elements to the left part of the array and all the large elements to the right part. "Small" and "large" are, of course, relative to the pivot.

While `i` is to the left of `j`, we move `i` right, skipping over elements that are smaller than the pivot. We move `j` left, skipping over elements that are larger than the pivot. When `i` and `j` have stopped, `i` is pointing at a large element and `j` is pointing at a small element. If

`i` is to the left of `j`, those elements are swapped. The effect is to push a large element to the right and a small element to the left. In the example above, `i` would not move and `j` would slide over one place. The situation is as follows:

8	1	4	9	0	3	5	2	7	6
↑							↑		
i							j		

We then swap the elements pointed to by `i` and `j` and repeat the process until `i` and `j` cross:

After First Swap

2	1	4	9	0	3	5	8	7	6
↑							↑		
i							j		

Before Second Swap

2	1	4	9	0	3	5	8	7	6
			↑			↑			
			i			j			

After Second Swap

2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			j			

Before Third Swap

2	1	4	5	0	3	9	8	7	6
					↑	↑			
					j	i			

At this stage, `i` and `j` have crossed, so no swap is performed. The final part of the partitioning is to swap the pivot element with the element pointed to by `i`:

After Swap with Pivot

2	1	4	5	0	3	6	8	7	9
						↑			↑
						i			pivot

When the pivot is swapped with `i` in the last step, we know that every element in a position $p <$ `i` must be small. This is because either position p contained a small element

to start with, or the large element originally in position p was replaced during a swap. A similar argument shows that elements in positions $p >$ `i` must be large.

One important detail we must consider is how to handle elements that are equal to the pivot. The questions are whether or not `i` should stop when it sees an element equal to the pivot and whether or not `j` should stop when it sees an element equal to the pivot. Intuitively, `i` and `j` ought to do the same thing, since otherwise the partitioning step is biased. For instance, if `i` stops and `j` does not, then all elements that are equal to the pivot will wind up in S_2.

To get an idea of what might be good, we consider the case where all the elements in the array are identical. If both `i` and `j` stop, there will be many swaps between identical elements. Although this seems useless, the positive effect is that `i` and `j` will cross in the middle, so when the pivot is replaced, the partition creates two nearly equal subarrays. The mergesort analysis tells us that the total running time would then be $O(N \log N)$.

If neither `i` nor `j` stops, and code is present to prevent them from running off the end of the array, no swaps will be performed. Although this seems good, a correct implementation would then swap the pivot into the last spot that `i` touched, which would be the next-to-last position (or last, depending on the exact implementation). This would create very uneven subarrays. If all the elements are identical, the running time is $O(N^2)$. The effect is the same as using the first element as a pivot for presorted input. It takes quadratic time to do nothing!

Thus, we find that it is better to do the unnecessary swaps and create even subarrays than to risk wildly uneven subarrays. Therefore, we will have both `i` and `j` stop if they encounter an element equal to the pivot. This turns out to be the only one of the four possibilities that does not take quadratic time for this input.

At first glance it may seem that worrying about an array of identical elements is silly. After all, why would anyone want to sort 500,000 identical elements? However, recall that quicksort is recursive. Suppose there are 10,000,000 elements, of which 500,000 are identical (or, more likely, complex elements whose sort keys are identical). Eventually, quicksort will make the recursive call on only these 500,000 elements. Then it really will be important to make sure that 500,000 identical elements can be sorted efficiently.

7.7.3 Small Arrays

For very small arrays ($N \leq 20$), quicksort does not perform as well as insertion sort. Furthermore, because quicksort is recursive, these cases will occur frequently. A common solution is not to use quicksort recursively for small arrays, but instead use a sorting algorithm that is efficient for small arrays, such as insertion sort. Using this strategy can actually save about 15 percent in the running time (over doing no cutoff at all). A good cutoff range is $N = 10$, although any cutoff between 5 and 20 is likely to produce similar results. This also saves nasty degenerate cases, such as taking the median of three elements when there are only one or two.

7.7.4 Actual Quicksort Routines

The driver for quicksort is shown in Figure 7.15.

```
1  /**
2   * Quicksort algorithm (driver).
3   */
4  template <typename Comparable>
5  void quicksort( vector<Comparable> & a )
6  {
7      quicksort( a, 0, a.size( ) - 1 );
8  }
```

Figure 7.15 Driver for quicksort

The general form of the routines will be to pass the array and the range of the array (`left` and `right`) to be sorted. The first routine to deal with is pivot selection. The easiest way to do this is to sort `a[left]`, `a[right]`, and `a[center]` in place. This has the extra advantage that the smallest of the three winds up in `a[left]`, which is where the partitioning step would put it anyway. The largest winds up in `a[right]`, which is also the correct place, since it is larger than the pivot. Therefore, we can place the pivot in `a[right - 1]` and initialize `i` and `j` to `left + 1` and `right - 2` in the partition phase. Yet another benefit is that because `a[left]` is smaller than the pivot, it will act as a sentinel for `j`. Thus, we do not need to worry about `j` running past the end. Since `i` will stop on elements equal to the pivot, storing the pivot in `a[right-1]` provides a sentinel for `i`. The code in

```
1  /**
2   * Return median of left, center, and right.
3   * Order these and hide the pivot.
4   */
5  template <typename Comparable>
6  const Comparable & median3( vector<Comparable> & a, int left, int right )
7  {
8      int center = ( left + right ) / 2;
9
10     if( a[ center ] < a[ left ] )
11         std::swap( a[ left ], a[ center ] );
12     if( a[ right ] < a[ left ] )
13         std::swap( a[ left ], a[ right ] );
14     if( a[ right ] < a[ center ] )
15         std::swap( a[ center ], a[ right ] );
16
17         // Place pivot at position right - 1
18     std::swap( a[ center ], a[ right - 1 ] );
19     return a[ right - 1 ];
20 }
```

Figure 7.16 Code to perform median-of-three partitioning

Figure 7.16 does the median-of-three partitioning with all the side effects described. It may seem that it is only slightly inefficient to compute the pivot by a method that does not actually sort `a[left]`, `a[center]`, and `a[right]`, but, surprisingly, this produces bad results (see Exercise 7.51).

The real heart of the quicksort routine is in Figure 7.17. It includes the partitioning and recursive calls. There are several things worth noting in this implementation. Line 16 initializes `i` and `j` to 1 past their correct values, so that there are no special cases to consider. This initialization depends on the fact that median-of-three partitioning has

```
1   /**
2    * Internal quicksort method that makes recursive calls.
3    * Uses median-of-three partitioning and a cutoff of 10.
4    * a is an array of Comparable items.
5    * left is the left-most index of the subarray.
6    * right is the right-most index of the subarray.
7    */
8   template <typename Comparable>
9   void quicksort( vector<Comparable> & a, int left, int right )
10  {
11      if( left + 10 <= right )
12      {
13          const Comparable & pivot = median3( a, left, right );
14
15              // Begin partitioning
16          int i = left, j = right - 1;
17          for( ; ; )
18          {
19              while( a[ ++i ] < pivot ) { }
20              while( pivot < a[ --j ] ) { }
21              if( i < j )
22                  std::swap( a[ i ], a[ j ] );
23              else
24                  break;
25          }
26
27          std::swap( a[ i ], a[ right - 1 ] );  // Restore pivot
28
29          quicksort( a, left, i - 1 );     // Sort small elements
30          quicksort( a, i + 1, right );    // Sort large elements
31      }
32      else  // Do an insertion sort on the subarray
33          insertionSort( a, left, right );
34  }
```

Figure 7.17 Main quicksort routine

```
16          int i = left + 1, j = right - 2;
17          for( ; ; )
18          {
19              while( a[ i ] < pivot ) i++;
20              while( pivot < a[ j ] ) j--;
21              if( i < j )
22                  std::swap( a[ i ], a[ j ] );
23              else
24                  break;
25          }
```

Figure 7.18 A small change to quicksort, which breaks the algorithm

some side effects; this program will not work if you try to use it without change with a simple pivoting strategy, because `i` and `j` start in the wrong place and there is no longer a sentinel for `j`.

The swapping action at line 22 is sometimes written explicitly, for speed purposes. For the algorithm to be fast, it is necessary to force the compiler to compile this code inline. Many compilers will do this automatically if `swap` is declared using `inline`, but for those that do not, the difference can be significant.

Finally, lines 19 and 20 show why quicksort is so fast. The inner loop of the algorithm consists of an increment/decrement (by 1, which is fast), a test, and a jump. There is no extra juggling as there is in mergesort. This code is still surprisingly tricky. It is tempting to replace lines 16 to 25 with the statements in Figure 7.18. This does not work, because there would be an infinite loop if `a[i] = a[j] = pivot`.

7.7.5 Analysis of Quicksort

Like mergesort, quicksort is recursive; therefore, its analysis requires solving a recurrence formula. We will do the analysis for a quicksort, assuming a random pivot (no median-of-three partitioning) and no cutoff for small arrays. We will take $T(0) = T(1) = 1$, as in mergesort. The running time of quicksort is equal to the running time of the two recursive calls plus the linear time spent in the partition (the pivot selection takes only constant time). This gives the basic quicksort relation

$$T(N) = T(\mathrm{i}) + T(N - i - 1) + cN \tag{7.1}$$

where $i = |S_1|$ is the number of elements in S_1. We will look at three cases.

Worst-Case Analysis

The pivot is the smallest element, all the time. Then $i = 0$, and if we ignore $T(0) = 1$, which is insignificant, the recurrence is

$$T(N) = T(N - 1) + cN, \quad N > 1 \tag{7.2}$$

We telescope, using Equation (7.2) repeatedly. Thus,

$$T(N-1) = T(N-2) + c(N-1) \tag{7.3}$$
$$T(N-2) = T(N-3) + c(N-2) \tag{7.4}$$
$$\vdots$$
$$T(2) = T(1) + c(2) \tag{7.5}$$

Adding up all these equations yields

$$T(N) = T(1) + c\sum_{i=2}^{N} i = \Theta(N^2) \tag{7.6}$$

as claimed earlier. To see that this is the worst possible case, note that the total cost of all the partitions in recursive calls at depth d must be at most N. Since the recursion depth is at most N, this gives an $O(N^2)$ worst-case bound for quicksort.

Best-Case Analysis

In the best case, the pivot is in the middle. To simplify the math, we assume that the two subarrays are each exactly half the size of the original, and although this gives a slight overestimate, this is acceptable because we are only interested in a Big-Oh answer.

$$T(N) = 2T(N/2) + cN \tag{7.7}$$

Divide both sides of Equation (7.7) by N.

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c \tag{7.8}$$

We will telescope using this equation:

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c \tag{7.9}$$
$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + c \tag{7.10}$$
$$\vdots$$
$$\frac{T(2)}{2} = \frac{T(1)}{1} + c \tag{7.11}$$

We add all the equations from (7.8) to (7.11) and note that there are $\log N$ of them:

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c\log N \tag{7.12}$$

which yields

$$T(N) = cN\log N + N = \Theta(N\log N) \tag{7.13}$$

Notice that this is the exact same analysis as mergesort; hence, we get the same answer. That this is the best case is implied by results in Section 7.8.

Average-Case Analysis

This is the most difficult part. For the average case, we assume that each of the sizes for S_1 is equally likely, and hence has probability $1/N$. This assumption is actually valid for our pivoting and partitioning strategy, but it is not valid for some others. Partitioning strategies that do not preserve the randomness of the subarrays cannot use this analysis. Interestingly, these strategies seem to result in programs that take longer to run in practice.

With this assumption, the average value of $T(i)$, and hence $T(N - i - 1)$, is $(1/N)\sum_{j=0}^{N-1} T(j)$. Equation (7.1) then becomes

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN \tag{7.14}$$

If Equation (7.14) is multiplied by N, it becomes

$$NT(N) = 2\left[\sum_{j=0}^{N-1} T(j)\right] + cN^2 \tag{7.15}$$

We need to remove the summation sign to simplify matters. We note that we can telescope with one more equation:

$$(N-1)T(N-1) = 2\left[\sum_{j=0}^{N-2} T(j)\right] + c(N-1)^2 \tag{7.16}$$

If we subtract Equation (7.16) from Equation (7.15), we obtain

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c \tag{7.17}$$

We rearrange terms and drop the insignificant $-c$ on the right, obtaining

$$NT(N) = (N+1)T(N-1) + 2cN \tag{7.18}$$

We now have a formula for $T(N)$ in terms of $T(N-1)$ only. Again the idea is to telescope, but Equation (7.18) is in the wrong form. Divide Equation (7.18) by $N(N+1)$:

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \tag{7.19}$$

Now we can telescope.

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N} \tag{7.20}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1} \tag{7.21}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \tag{7.22}$$

Adding Equations (7.19) through (7.22) yields

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i} \tag{7.23}$$

The sum is about $\log_e(N+1) + \gamma - \frac{3}{2}$, where $\gamma \approx 0.577$ is known as Euler's constant, so

$$\frac{T(N)}{N+1} = O(\log N) \tag{7.24}$$

And so

$$T(N) = O(N \log N) \tag{7.25}$$

Although this analysis seems complicated, it really is not—the steps are natural once you have seen some recurrence relations. The analysis can actually be taken further. The highly optimized version that was described above has also been analyzed, and this result gets extremely difficult, involving complicated recurrences and advanced mathematics. The effect of equal elements has also been analyzed in detail, and it turns out that the code presented does the right thing.

7.7.6 A Linear-Expected-Time Algorithm for Selection

Quicksort can be modified to solve the *selection problem,* which we have seen in Chapters 1 and 6. Recall that by using a priority queue, we can find the kth largest (or smallest) element in $O(N + k \log N)$. For the special case of finding the median, this gives an $O(N \log N)$ algorithm.

Since we can sort the array in $O(N \log N)$ time, one might expect to obtain a better time bound for selection. The algorithm we present to find the kth smallest element in a set S is almost identical to quicksort. In fact, the first three steps are the same. We will call this algorithm **quickselect.** Let $|S_i|$ denote the number of elements in S_i. The steps of quickselect are

1. If $|S| = 1$, then $k = 1$ and return the element in S as the answer. If a cutoff for small arrays is being used and $|S| \leq$ `CUTOFF`, then sort S and return the kth smallest element.
2. Pick a pivot element, $v \in S$.
3. Partition $S - \{v\}$ into S_1 and S_2, as was done with quicksort.
4. If $k \leq |S_1|$, then the kth smallest element must be in S_1. In this case, return quickselect(S_1, k). If $k = 1 + |S_1|$, then the pivot is the kth smallest element and we can return it as the answer. Otherwise, the kth smallest element lies in S_2, and it is the $(k - |S_1| - 1)$st smallest element in S_2. We make a recursive call and return quickselect$(S_2, k - |S_1| - 1)$.

In contrast to quicksort, quickselect makes only one recursive call instead of two. The worst case of quickselect is identical to that of quicksort and is $O(N^2)$. Intuitively, this is because quicksort's worst case is when one of S_1 and S_2 is empty; thus, quickselect is not

really saving a recursive call. The average running time, however, is $O(N)$. The analysis is similar to quicksort's and is left as an exercise.

The implementation of quickselect is even simpler than the abstract description might imply. The code to do this is shown in Figure 7.19. When the algorithm terminates, the

```
1   /**
2    * Internal selection method that makes recursive calls.
3    * Uses median-of-three partitioning and a cutoff of 10.
4    * Places the kth smallest item in a[k-1].
5    * a is an array of Comparable items.
6    * left is the left-most index of the subarray.
7    * right is the right-most index of the subarray.
8    * k is the desired rank (1 is minimum) in the entire array.
9    */
10  template <typename Comparable>
11  void quickSelect( vector<Comparable> & a, int left, int right, int k )
12  {
13      if( left + 10 <= right )
14      {
15          const Comparable & pivot = median3( a, left, right );
16
17              // Begin partitioning
18          int i = left, j = right - 1;
19          for( ; ; )
20          {
21              while( a[ ++i ] < pivot ) { }
22              while( pivot < a[ --j ] ) { }
23              if( i < j )
24                  std::swap( a[ i ], a[ j ] );
25              else
26                  break;
27          }
28
29          std::swap( a[ i ], a[ right - 1 ] );  // Restore pivot
30
31              // Recurse; only this part changes
32          if( k <= i )
33              quickSelect( a, left, i - 1, k );
34          else if( k > i + 1 )
35              quickSelect( a, i + 1, right, k );
36      }
37      else  // Do an insertion sort on the subarray
38          insertionSort( a, left, right );
39  }
```

Figure 7.19 Main quickselect routine

kth smallest element is in position $k-1$ (because arrays start at index 0). This destroys the original ordering; if this is not desirable, then a copy must be made.

Using a median-of-three pivoting strategy makes the chance of the worst case occurring almost negligible. By carefully choosing the pivot, however, we can eliminate the quadratic worst case and ensure an $O(N)$ algorithm. The overhead involved in doing this is considerable, so the resulting algorithm is mostly of theoretical interest. In Chapter 10, we will examine the linear-time worst-case algorithm for selection, and we shall also see an interesting technique of choosing the pivot that results in a somewhat faster selection algorithm in practice.

7.8 A General Lower Bound for Sorting

Although we have $O(N \log N)$ algorithms for sorting, it is not clear that this is as good as we can do. In this section, we prove that any algorithm for sorting that uses only comparisons requires $\Omega(N \log N)$ comparisons (and hence time) in the worst case, so that mergesort and heapsort are optimal to within a constant factor. The proof can be extended to show that $\Omega(N \log N)$ comparisons are required, even on average, for any sorting algorithm that uses only comparisons, which means that quicksort is optimal on average to within a constant factor.

Specifically, we will prove the following result: Any sorting algorithm that uses only comparisons requires $\lceil \log(N!) \rceil$ comparisons in the worst case and $\log(N!)$ comparisons on average. We will assume that all N elements are distinct, since any sorting algorithm must work for this case.

7.8.1 Decision Trees

A **decision tree** is an abstraction used to prove lower bounds. In our context, a decision tree is a binary tree. Each node represents a set of possible orderings, consistent with comparisons that have been made, among the elements. The results of the comparisons are the tree edges.

The decision tree in Figure 7.20 represents an algorithm that sorts the three elements a, b, and c. The initial state of the algorithm is at the root. (We will use the terms *state* and *node* interchangeably.) No comparisons have been done, so all orderings are legal. The first comparison that *this particular* algorithm performs compares a and b. The two results lead to two possible states. If $a < b$, then only three possibilities remain. If the algorithm reaches node 2, then it will compare a and c. Other algorithms might do different things; a different algorithm would have a different decision tree. If $a > c$, the algorithm enters state 5. Since there is only one ordering that is consistent, the algorithm can terminate and report that it has completed the sort. If $a < c$, the algorithm cannot do this, because there are two possible orderings and it cannot possibly be sure which is correct. In this case, the algorithm will require one more comparison.

Every algorithm that sorts by using only comparisons can be represented by a decision tree. Of course, it is only feasible to draw the tree for extremely small input sizes. The number of comparisons used by the sorting algorithm is equal to the depth of the deepest