CHAPTER 7

Sorting

In this chapter, we discuss the problem of sorting an array of elements. To simplify matters, we will assume in our examples that the array contains only integers, although our code will once again allow more general objects. For most of this chapter, we will also assume that the entire sort can be done in main memory, so that the number of elements is relatively small (less than a few million). Sorts that cannot be performed in main memory and must be done on disk or tape are also quite important. This type of sorting, known as external sorting, will be discussed at the end of the chapter.

Our investigation of internal sorting will show that. . .

- There are several easy algorithms to sort in $O(N^2)$, such as insertion sort.
- There is an algorithm, Shellsort, that is very simple to code, runs in $o(N^2)$, and is efficient in practice.
- There are slightly more complicated $O(N \log N)$ sorting algorithms.
- Any general-purpose sorting algorithm requires $\Omega(N \log N)$ comparisons.

The rest of this chapter will describe and analyze the various sorting algorithms. These algorithms contain interesting and important ideas for code optimization as well as algorithm design. Sorting is also an example where the analysis can be precisely performed. Be forewarned that where appropriate, we will do as much analysis as possible.

7.1 Preliminaries

The algorithms we describe will all be interchangeable. Each will be passed an array containing the elements; we assume all array positions contain data to be sorted. We will assume that N is the number of elements passed to our sorting routines.

We will also assume the existence of the "<" and ">" operators, which can be used to place a consistent ordering on the input. Besides the assignment operator, these are the only operations allowed on the input data. Sorting under these conditions is known as **comparison-based sorting**.

This interface is not the same as in the STL sorting algorithms. In the STL, sorting is accomplished by use of the function template `sort`. The parameters to `sort` represent the start and endmarker of a (range in a) container and an optional comparator:

```
void sort( Iterator begin, Iterator end );
void sort( Iterator begin, Iterator end, Comparator cmp );
```

The iterators must support random access. The `sort` algorithm does not guarantee that equal items retain their original order (if that is important, use `stable_sort` instead of `sort`).

As an example, in

```
std::sort( v.begin( ), v.end( ) );
std::sort( v.begin( ), v.end( ), greater<int>{ } );
std::sort( v.begin( ), v.begin( ) + ( v.end( ) - v.begin( ) ) / 2 );
```

the first call sorts the entire container, `v`, in nondecreasing order. The second call sorts the entire container in nonincreasing order. The third call sorts the first half of the container in nondecreasing order.

The sorting algorithm used is generally quicksort, which we describe in Section 7.7. In Section 7.2, we implement the simplest sorting algorithm using both our style of passing the array of comparable items, which yields the most straightforward code, and the interface supported by the STL, which requires more code.

7.2 Insertion Sort

One of the simplest sorting algorithms is the **insertion sort**.

7.2.1 The Algorithm

Insertion sort consists of $N-1$ **passes.** For pass $p=1$ through $N-1$, insertion sort ensures that the elements in positions 0 through p are in sorted order. Insertion sort makes use of the fact that elements in positions 0 through $p-1$ are already known to be in sorted order. Figure 7.1 shows a sample array after each pass of insertion sort.

Figure 7.1 shows the general strategy. In pass p, we move the element in position p left until its correct place is found among the first $p+1$ elements. The code in Figure 7.2 implements this strategy. Lines 11 to 14 implement that data movement without the explicit use of swaps. The element in position p is moved to `tmp`, and all larger elements (prior to position p) are moved one spot to the right. Then `tmp` is moved to the correct spot. This is the same technique that was used in the implementation of binary heaps.

Original	34	8	64	51	32	21	Positions Moved
After $p=1$	8	34	64	51	32	21	1
After $p=2$	8	34	64	51	32	21	0
After $p=3$	8	34	51	64	32	21	1
After $p=4$	8	32	34	51	64	21	3
After $p=5$	8	21	32	34	51	64	4

Figure 7.1 Insertion sort after each pass

```
1   /**
2    * Simple insertion sort.
3    */
4   template <typename Comparable>
5   void insertionSort( vector<Comparable> & a )
6   {
7       for( int p = 1; p < a.size( ); ++p )
8       {
9           Comparable tmp = std::move( a[ p ] );
10
11          int j;
12          for( j = p; j > 0 && tmp < a[ j - 1 ]; --j )
13              a[ j ] = std::move( a[ j - 1 ] );
14          a[ j ] = std::move( tmp );
15      }
16  }
```

Figure 7.2 Insertion sort routine

7.2.2 STL Implementation of Insertion Sort

In the STL, instead of having the sort routines take an array of comparable items as a single parameter, the sort routines receive a pair of iterators that represent the start and endmarker of a range. A two-parameter sort routine uses just that pair of iterators and presumes that the items can be ordered, while a three-parameter sort routine has a function object as a third parameter.

Converting the algorithm in Figure 7.2 to use the STL introduces several issues. The obvious issues are

1. We must write a two-parameter sort and a three-parameter sort. Presumably, the two-parameter sort invokes the three-parameter sort, with `less<Object>{ }` as the third parameter.
2. Array access must be converted to iterator access.
3. Line 11 of the original code requires that we create `tmp`, which in the new code will have type `Object`.

The first issue is the trickiest because the template type parameters (i.e., the generic types) for the two-parameter sort are both `Iterator`; however, `Object` is not one of the generic type parameters. Prior to C++11, one had to write extra routines to solve this problem. As shown in Figure 7.3, C++11 introduces `decltype` which cleanly expresses the intent.

Figure 7.4 shows the main sorting code that replaces array indexing with use of the iterator, and that replaces calls to `operator<` with calls to the `lessThan` function object.

Observe that once we actually code the `insertionSort` algorithm, every statement in the original code is replaced with a corresponding statement in the new code that makes

```
/*
 * The two-parameter version calls the three-parameter version,
 * using C++11 decltype
 */
template <typename Iterator>
void insertionSort( const Iterator & begin, const Iterator & end )
{
    insertionSort( begin, end, less<decltype(*begin)>{ } );
}
```

Figure 7.3 Two-parameter sort invokes three-parameter sort via C++11 `decltype`

```
template <typename Iterator, typename Comparator>
void insertionSort( const Iterator & begin, const Iterator & end,
                    Comparator lessThan )
{
    if( begin == end )
        return;

    Iterator j;

    for( Iterator p = begin+1; p != end; ++p )
    {
        auto tmp = std::move( *p );
        for( j = p; j != begin && lessThan( tmp, *( j-1 ) ); --j )
            *j = std::move( *(j-1) );
        *j = std::move( tmp );
    }
}
```

Figure 7.4 Three-parameter sort using iterators

straightforward use of iterators and the function object. The original code is arguably much simpler to read, which is why we use our simpler interface rather than the STL interface when coding our sorting algorithms.

7.2.3 Analysis of Insertion Sort

Because of the nested loops, each of which can take N iterations, insertion sort is $O(N^2)$. Furthermore, this bound is tight, because input in reverse order can achieve this bound. A precise calculation shows that the number of tests in the inner loop in Figure 7.2 is at most $p + 1$ for each value of p. Summing over all p gives a total of

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$

On the other hand, if the input is presorted, the running time is $O(N)$, because the test in the inner `for` loop always fails immediately. Indeed, if the input is almost sorted (this term will be more rigorously defined in the next section), insertion sort will run quickly. Because of this wide variation, it is worth analyzing the average-case behavior of this algorithm. It turns out that the average case is $\Theta(N^2)$ for insertion sort, as well as for a variety of other sorting algorithms, as the next section shows.

7.3 A Lower Bound for Simple Sorting Algorithms

An **inversion** in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$. In the example of the last section, the input list 34, 8, 64, 51, 32, 21 had nine inversions, namely $(34, 8)$, $(34, 32)$, $(34, 21)$, $(64, 51)$, $(64, 32)$, $(64, 21)$, $(51, 32)$, $(51, 21)$, and $(32, 21)$. Notice that this is exactly the number of swaps that needed to be (implicitly) performed by insertion sort. This is always the case, because swapping two adjacent elements that are out of place removes exactly one inversion, and a sorted array has no inversions. Since there is $O(N)$ other work involved in the algorithm, the running time of insertion sort is $O(I + N)$, where I is the number of inversions in the original array. Thus, insertion sort runs in linear time if the number of inversions is $O(N)$.

We can compute precise bounds on the average running time of insertion sort by computing the average number of inversions in a permutation. As usual, defining *average* is a difficult proposition. We will assume that there are no duplicate elements (if we allow duplicates, it is not even clear what the average number of duplicates is). Using this assumption, we can assume that the input is some permutation of the first N integers (since only relative ordering is important) and that all are equally likely. Under these assumptions, we have the following theorem:

Theorem 7.1
The average number of inversions in an array of N distinct elements is $N(N-1)/4$.

Proof
For any list, L, of elements, consider L_r, the list in reverse order. The reverse list of the example is $21, 32, 51, 64, 8, 34$. Consider any pair of two elements in the list (x, y) with $y > x$. Clearly, in exactly one of L and L_r this ordered pair represents an inversion. The total number of these pairs in a list L and its reverse L_r is $N(N-1)/2$. Thus, an average list has half this amount, or $N(N-1)/4$ inversions.

This theorem implies that insertion sort is quadratic on average. It also provides a very strong lower bound about any algorithm that only exchanges adjacent elements.

Theorem 7.2
Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average.

Proof

The average number of inversions is initially $N(N-1)/4 = \Omega(N^2)$. Each swap removes only one inversion, so $\Omega(N^2)$ swaps are required.

This is an example of a lower-bound proof. It is valid not only for insertion sort, which performs adjacent exchanges implicitly, but also for other simple algorithms such as bubble sort and selection sort, which we will not describe here. In fact, it is valid over an entire *class* of sorting algorithms, including those undiscovered, that perform only adjacent exchanges. Because of this, this proof cannot be confirmed empirically. Although this lower-bound proof is rather simple, in general proving lower bounds is much more complicated than proving upper bounds and in some cases resembles magic.

This lower bound shows us that in order for a sorting algorithm to run in subquadratic, or $o(N^2)$, time, it must do comparisons and, in particular, exchanges between elements that are far apart. A sorting algorithm makes progress by eliminating inversions, and to run efficiently, it must eliminate more than just one inversion per exchange.

7.4 Shellsort

Shellsort, named after its inventor, Donald Shell, was one of the first algorithms to break the quadratic time barrier, although it was not until several years after its initial discovery that a subquadratic time bound was proven. As suggested in the previous section, it works by comparing elements that are distant; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared. For this reason, Shellsort is sometimes referred to as **diminishing increment sort**.

Shellsort uses a sequence, $h_1, h_2, \ldots, h_t$, called the **increment sequence**. Any increment sequence will do as long as $h_1 = 1$, but some choices are better than others (we will discuss that issue later). After a *phase,* using some increment h_k, for every i, we have $a[i] \leq a[i + h_k]$ (where this makes sense); all elements spaced h_k apart are sorted. The file is then said to be **h_k-sorted.** For example, Figure 7.5 shows an array after several phases of Shellsort. An important property of Shellsort (which we state without proof) is that an h_k-sorted file that is then h_{k-1}-sorted remains h_k-sorted. If this were not the case, the algorithm would likely be of little value, since work done by early phases would be undone by later phases.

The general strategy to h_k-sort is for each position, i, in $h_k, h_k + 1, \ldots, N - 1$, place the element in the correct spot among $i, i - h_k, i - 2h_k$, and so on. Although this does not

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Figure 7.5 Shellsort after each pass

```
1   /**
2    * Shellsort, using Shell's (poor) increments.
3    */
4   template <typename Comparable>
5   void shellsort( vector<Comparable> & a )
6   {
7       for( int gap = a.size( ) / 2; gap > 0; gap /= 2 )
8           for( int i = gap; i < a.size( ); ++i )
9           {
10              Comparable tmp = std::move( a[ i ] );
11              int j = i;
12
13              for( ; j >= gap && tmp < a[ j - gap ]; j -= gap )
14                  a[ j ] = std::move( a[ j - gap ] );
15              a[ j ] = std::move( tmp );
16          }
17  }
```

Figure 7.6 Shellsort routine using Shell's increments (better increments are possible)

affect the implementation, a careful examination shows that the action of an h_k-sort is to perform an insertion sort on h_k independent subarrays. This observation will be important when we analyze the running time of Shellsort.

A popular (but poor) choice for increment sequence is to use the sequence suggested by Shell: $h_t = \lfloor N/2 \rfloor$, and $h_k = \lfloor h_{k+1}/2 \rfloor$. Figure 7.6 contains a function that implements Shellsort using this sequence. We shall see later that there are increment sequences that give a significant improvement in the algorithm's running time; even a minor change can drastically affect performance (Exercise 7.10).

The program in Figure 7.6 avoids the explicit use of swaps in the same manner as our implementation of insertion sort.

7.4.1 Worst-Case Analysis of Shellsort

Although Shellsort is simple to code, the analysis of its running time is quite another story. The running time of Shellsort depends on the choice of increment sequence, and the proofs can be rather involved. The average-case analysis of Shellsort is a long-standing open problem, except for the most trivial increment sequences. We will prove tight worst-case bounds for two particular increment sequences.

Theorem 7.3

The worst-case running time of Shellsort using Shell's increments is $\Theta(N^2)$.

Proof

The proof requires showing not only an upper bound on the worst-case running time but also showing that there exists some input that actually takes $\Omega(N^2)$ time to run.

We prove the lower bound first by constructing a bad case. First, we choose N to be a power of 2. This makes all the increments even, except for the last increment, which is 1. Now, we will give as input an array with the $N/2$ largest numbers in the even positions and the $N/2$ smallest numbers in the odd positions (for this proof, the first position is position 1). As all the increments except the last are even, when we come to the last pass, the $N/2$ largest numbers are still all in even positions and the $N/2$ smallest numbers are still all in odd positions. The ith smallest number ($i \leq N/2$) is thus in position $2i - 1$ before the beginning of the last pass. Restoring the ith element to its correct place requires moving it $i-1$ spaces in the array. Thus, to merely place the $N/2$ smallest elements in the correct place requires at least $\sum_{i=1}^{N/2} i - 1 = \Omega(N^2)$ work. As an example, Figure 7.7 shows a bad (but not the worst) input when $N = 16$. The number of inversions remaining after the 2-sort is exactly $1+2+3+4+5+6+7 = 28$; thus, the last pass will take considerable time.

To finish the proof, we show the upper bound of $O(N^2)$. As we have observed before, a pass with increment h_k consists of h_k insertion sorts of about N/h_k elements. Since insertion sort is quadratic, the total cost of a pass is $O(h_k(N/h_k)^2) = O(N^2/h_k)$. Summing over all passes gives a total bound of $O(\sum_{i=1}^{t} N^2/h_i) = O(N^2 \sum_{i=1}^{t} 1/h_i)$. Because the increments form a geometric series with common ratio 2, and the largest term in the series is $h_1 = 1$, $\sum_{i=1}^{t} 1/h_i < 2$. Thus we obtain a total bound of $O(N^2)$.

The problem with Shell's increments is that pairs of increments are not necessarily relatively prime, and thus the smaller increment can have little effect. Hibbard suggested a slightly different increment sequence, which gives better results in practice (and theoretically). His increments are of the form $1, 3, 7, \ldots, 2^k - 1$. Although these increments are almost identical, the key difference is that consecutive increments have no common factors. We now analyze the worst-case running time of Shellsort for this increment sequence. The proof is rather complicated.

Theorem 7.4

The worst-case running time of Shellsort using Hibbard's increments is $\Theta(N^{3/2})$.

Proof

We will prove only the upper bound and leave the proof of the lower bound as an exercise. The proof requires some well-known results from additive number theory. References to these results are provided at the end of the chapter.

For the upper bound, as before, we bound the running time of each pass and sum over all passes. For increments $h_k > N^{1/2}$, we will use the bound $O(N^2/h_k)$ from the

Start	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 7.7 Bad case for Shellsort with Shell's increments (positions are numbered 1 to 16)

previous theorem. Although this bound holds for the other increments, it is too large to be useful. Intuitively, we must take advantage of the fact that *this* increment sequence is *special*. What we need to show is that for any element $a[p]$ in position p, when it is time to perform an h_k-sort, there are only a few elements to the left of position p that are larger than $a[p]$.

When we come to h_k-sort the input array, we know that it has already been h_{k+1}- and h_{k+2}-sorted. Prior to the h_k-sort, consider elements in positions p and $p - i$, $i \leq p$. If i is a multiple of h_{k+1} or h_{k+2}, then clearly $a[p - i] < a[p]$. We can say more, however. If i is expressible as a linear combination (in nonnegative integers) of h_{k+1} and h_{k+2}, then $a[p - i] < a[p]$. As an example, when we come to 3-sort, the file is already 7- and 15-sorted. 52 is expressible as a linear combination of 7 and 15, because $52 = 1 * 7 + 3 * 15$. Thus, $a[100]$ cannot be larger than $a[152]$ because $a[100] \leq a[107] \leq a[122] \leq a[137] \leq a[152]$.

Now, $h_{k+2} = 2h_{k+1} + 1$, so h_{k+1} and h_{k+2} cannot share a common factor. In this case, it is possible to show that all integers that are at least as large as $(h_{k+1} - 1)(h_{k+2} - 1) = 8h_k^2 + 4h_k$ can be expressed as a linear combination of h_{k+1} and h_{k+2} (see the reference at the end of the chapter).

This tells us that the body of the innermost `for` loop can be executed at most $8h_k + 4 = O(h_k)$ times for each of the $N - h_k$ positions. This gives a bound of $O(Nh_k)$ per pass.

Using the fact that about half the increments satisfy $h_k < \sqrt{N}$, and assuming that t is even, the total running time is then

$$O\left(\sum_{k=1}^{t/2} Nh_k + \sum_{k=t/2+1}^{t} N^2/h_k\right) = O\left(N\sum_{k=1}^{t/2} h_k + N^2 \sum_{k=t/2+1}^{t} 1/h_k\right)$$

Because both sums are geometric series, and since $h_{t/2} = \Theta(\sqrt{N})$, this simplifies to

$$= O\left(Nh_{t/2}\right) + O\left(\frac{N^2}{h_{t/2}}\right) = O(N^{3/2})$$

The average-case running time of Shellsort, using Hibbard's increments, is thought to be $O(N^{5/4})$, based on simulations, but nobody has been able to prove this. Pratt has shown that the $\Theta(N^{3/2})$ bound applies to a wide range of increment sequences.

Sedgewick has proposed several increment sequences that give an $O(N^{4/3})$ worst-case running time (also achievable). The average running time is conjectured to be $O(N^{7/6})$ for these increment sequences. Empirical studies show that these sequences perform significantly better in practice than Hibbard's. The best of these is the sequence $\{1, 5, 19, 41, 109, \ldots\}$, in which the terms are either of the form $9 \cdot 4^i - 9 \cdot 2^i + 1$ or $4^i - 3 \cdot 2^i + 1$. This is most easily implemented by placing these values in an array. This increment sequence is the best known in practice, although there is a lingering possibility that some increment sequence might exist that could give a significant improvement in the running time of Shellsort.

There are several other results on Shellsort that (generally) require difficult theorems from number theory and combinatorics and are mainly of theoretical interest. Shellsort is a fine example of a very simple algorithm with an extremely complex analysis.

The performance of Shellsort is quite acceptable in practice, even for N in the tens of thousands. The simplicity of the code makes it the algorithm of choice for sorting up to moderately large input.

7.5 Heapsort

As mentioned in Chapter 6, priority queues can be used to sort in $O(N \log N)$ time. The algorithm based on this idea is known as **heapsort** and gives the best Big-Oh running time we have seen so far.

Recall from Chapter 6 that the basic strategy is to build a binary heap of N elements. This stage takes $O(N)$ time. We then perform N `deleteMin` operations. The elements leave the heap smallest first, in sorted order. By recording these elements in a second array and then copying the array back, we sort N elements. Since each `deleteMin` takes $O(\log N)$ time, the total running time is $O(N \log N)$.

The main problem with this algorithm is that it uses an extra array. Thus, the memory requirement is doubled. This could be a problem in some instances. Notice that the extra time spent copying the second array back to the first is only $O(N)$, so that this is not likely to affect the running *time* significantly. The problem is space.

A clever way to avoid using a second array makes use of the fact that after each `deleteMin`, the heap shrinks by 1. Thus the cell that was last in the heap can be used to store the element that was just deleted. As an example, suppose we have a heap with six elements. The first `deleteMin` produces a_1. Now the heap has only five elements, so we can place a_1 in position 6. The next `deleteMin` produces a_2. Since the heap will now only have four elements, we can place a_2 in position 5.

Using this strategy, after the last `deleteMin` the array will contain the elements in *decreasing* sorted order. If we want the elements in the more typical *increasing* sorted order, we can change the ordering property so that the parent has a larger element than the child. Thus, we have a (*max*)heap.

In our implementation, we will use a (*max*)heap but avoid the actual ADT for the purposes of speed. As usual, everything is done in an array. The first step builds the heap in linear time. We then perform $N - 1$ `deleteMax`es by swapping the last element in the heap with the first, decrementing the heap size, and percolating down. When the algorithm terminates, the array contains the elements in sorted order. For instance, consider the input sequence $31, 41, 59, 26, 53, 58, 97$. The resulting heap is shown in Figure 7.8.

Figure 7.9 shows the heap that results after the first `deleteMax`. As the figures imply, the last element in the heap is 31; 97 has been placed in a part of the heap array that is technically no longer part of the heap. After 5 more `deleteMax` operations, the heap will actually have only one element, but the elements left in the heap array will be in sorted order.

The code to perform heapsort is given in Figure 7.10. The slight complication is that, unlike the binary heap, where the data begin at array index 1, the array for heapsort contains data in position 0. Thus the code is a little different from the binary heap code. The changes are minor.

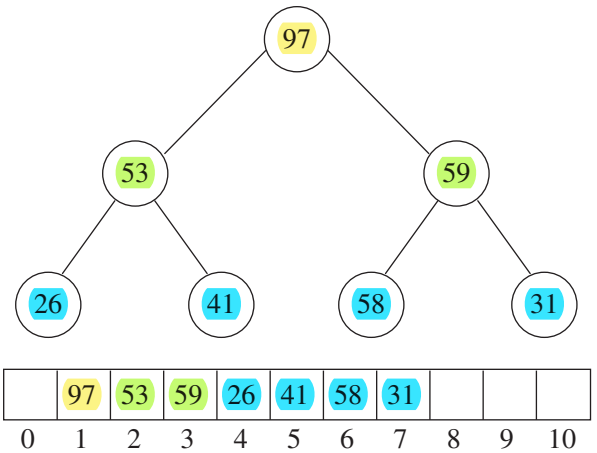


Figure 7.8 (*Max*) heap after `buildHeap` phase

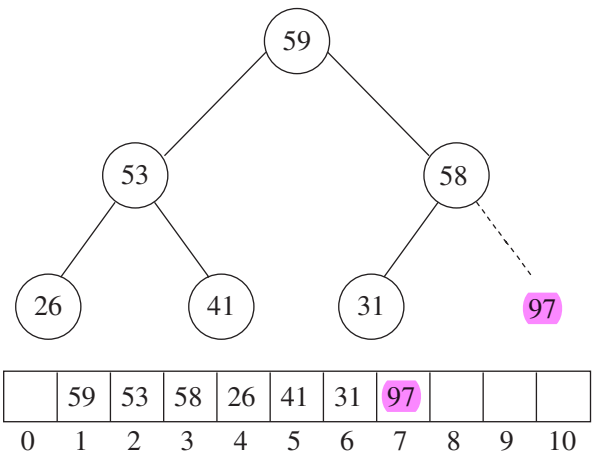


Figure 7.9 Heap after first `deleteMax`

7.5.1 Analysis of Heapsort

As we saw in Chapter 6, the first phase, which constitutes the building of the heap, uses less than $2N$ comparisons. In the second phase, the ith `deleteMax` uses at most less than $2\lfloor \log(N - i + 1) \rfloor$ comparisons, for a total of at most $2N \log N - O(N)$ comparisons (assuming $N \geq 2$). Consequently, in the worst case, at most $2N \log N - O(N)$ comparisons are used by heapsort. Exercise 7.13 asks you to show that it is possible for all of the `deleteMax` operations to achieve their worst case simultaneously.

```
/**
 * Standard heapsort.
 */
template <typename Comparable>
void heapsort( vector<Comparable> & a )
{
    for( int i = a.size( ) / 2 - 1; i >= 0; --i )  /* buildHeap */
        percDown( a, i, a.size( ) );
    for( int j = a.size( ) - 1; j > 0; --j )
    {
        std::swap( a[ 0 ], a[ j ] );               /* deleteMax */
        percDown( a, 0, j );
    }
}

/**
 * Internal method for heapsort.
 * i is the index of an item in the heap.
 * Returns the index of the left child.
 */
inline int leftChild( int i )
{
    return 2 * i + 1;
}

/**
 * Internal method for heapsort that is used in deleteMax and buildHeap.
 * i is the position from which to percolate down.
 * n is the logical size of the binary heap.
 */
template <typename Comparable>
void percDown( vector<Comparable> & a, int i, int n )
{
    int child;
    Comparable tmp;

    for( tmp = std::move( a[ i ] ); leftChild( i ) < n; i = child )
    {
        child = leftChild( i );
        if( child != n - 1 && a[ child ] < a[ child + 1 ] )
            ++child;
        if( tmp < a[ child ] )
            a[ i ] = std::move( a[ child ] );
        else
            break;
    }
    a[ i ] = std::move( tmp );
}
```

Figure 7.10 Heapsort

Experiments have shown that the performance of heapsort is extremely consistent: On average it uses only slightly fewer comparisons than the worst-case bound suggests. For many years, nobody had been able to show nontrivial bounds on heapsort's average running time. The problem, it seems, is that successive `deleteMax` operations destroy the heap's randomness, making the probability arguments very complex. Eventually, another approach proved successful.

Theorem 7.5

The average number of comparisons used to heapsort a random permutation of N distinct items is $2N \log N - O(N \log \log N)$.

Proof

The heap construction phase uses $\Theta(N)$ comparisons on average, and so we only need to prove the bound for the second phase. We assume a permutation of $\{1, 2, \ldots, N\}$.

Suppose the ith `deleteMax` pushes the root element down d_i levels. Then it uses $2d_i$ comparisons. For heapsort on any input, there is a cost sequence $D : d_1, d_2, \ldots, d_N$ that defines the cost of phase 2. That cost is given by $M_D = \sum_{i=1}^{N} d_i$; the number of comparisons used is thus $2M_D$.

Let $f(N)$ be the number of heaps of N items. One can show (Exercise 7.58) that $f(N) > (N/(4e))^N$ (where $e = 2.71828\ldots$). We will show that only an exponentially small fraction of these heaps (in particular $(N/16)^N$) have a cost smaller than $M = N(\log N - \log \log N - 4)$. When this is shown, it follows that the average value of M_D is at least M minus a term that is $o(1)$, and thus the average number of comparisons is at least $2M$. Consequently, our basic goal is to show that there are very few heaps that have small cost sequences.

Because level d_i has at most 2^{d_i} nodes, there are 2^{d_i} possible places that the root element can go for any d_i. Consequently, for any sequence D, the number of distinct corresponding `deleteMax` sequences is at most

$$S_D = 2^{d_1} 2^{d_2} \cdots 2^{d_N}$$

A simple algebraic manipulation shows that *for a given sequence* D,

$$S_D = 2^{M_D}$$

Because each d_i can assume any value between 1 and $\lfloor \log N \rfloor$, there are at most $(\log N)^N$ possible sequences D. It follows that the number of distinct `deleteMax` sequences that require cost exactly equal to M is at most the number of cost sequences of total cost M times the number of `deleteMax` sequences for each of these cost sequences. A bound of $(\log N)^N 2^M$ follows immediately.

The total number of heaps with cost sequence less than M is at most

$$\sum_{i=1}^{M-1} (\log N)^N 2^i < (\log N)^N 2^M$$

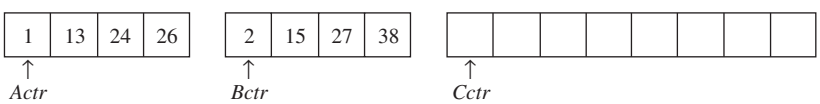
If we choose $M = N(\log N - \log\log N - 4)$, then the number of heaps that have cost sequence less than M is at most $(N/16)^N$, and the theorem follows from our earlier comments.

Using a more complex argument, it can be shown that heapsort always uses at least $N \log N - O(N)$ comparisons and that there are inputs that can achieve this bound. The average-case analysis also can be improved to $2N \log N - O(N)$ comparisons (rather than the nonlinear second term in Theorem 7.5).

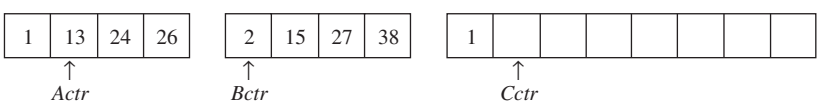
7.6 Mergesort

We now turn our attention to **mergesort**. Mergesort runs in $O(N \log N)$ worst-case running time, and the number of comparisons used is nearly optimal. It is a fine example of a recursive algorithm.

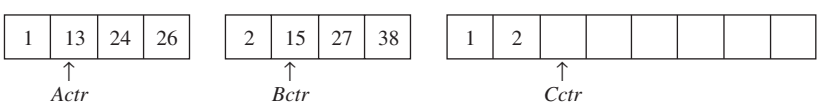
The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list. The basic merging algorithm takes two input arrays A and B, an output array C, and three counters, *Actr*, *Bctr*, and *Cctr*, which are initially set to the beginning of their respective arrays. The smaller of $A[Actr]$ and $B[Bctr]$ is copied to the next entry in C, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to C. An example of how the merge routine works is provided for the following input.



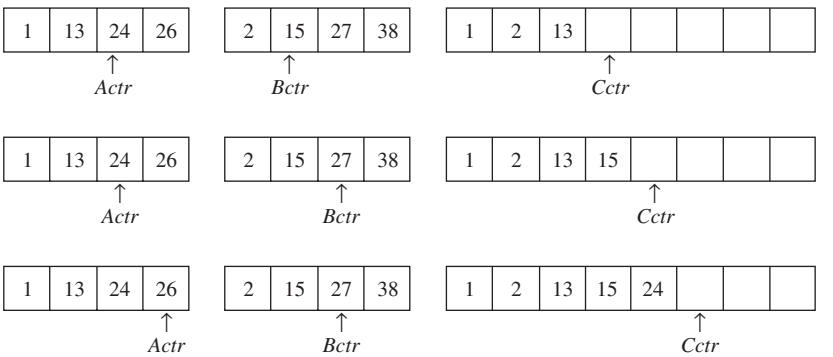
If the array A contains 1, 13, 24, 26, and B contains 2, 15, 27, 38, then the algorithm proceeds as follows: First, a comparison is done between 1 and 2. 1 is added to C, and then 13 and 2 are compared.



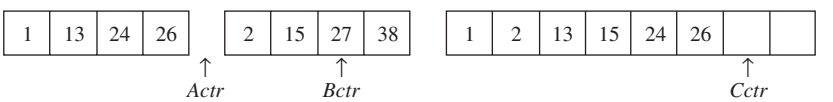
2 is added to C, and then 13 and 15 are compared.



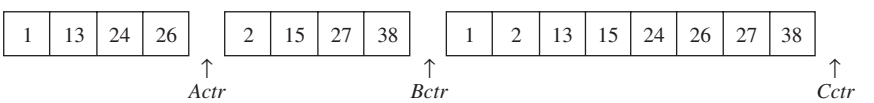
13 is added to *C*, and then 24 and 15 are compared. This proceeds until 26 and 27 are compared.



26 is added to *C*, and the *A* array is exhausted.



The remainder of the *B* array is then copied to *C*.



The time to merge two sorted lists is clearly linear, because at most $N - 1$ comparisons are made, where N is the total number of elements. To see this, note that every comparison adds an element to *C*, except the last comparison, which adds at least two.

The mergesort algorithm is therefore easy to describe. If $N = 1$, there is only one element to sort, and the answer is at hand. Otherwise, recursively mergesort the first half and the second half. This gives two sorted halves, which can then be merged together using the merging algorithm described above. For instance, to sort the eight-element array 24, 13, 26, 1, 2, 27, 38, 15, we recursively sort the first four and last four elements, obtaining 1, 13, 24, 26, 2, 15, 27, 38. Then we merge the two halves as above, obtaining the final list 1, 2, 13, 15, 24, 26, 27, 38. This algorithm is a classic divide-and-conquer strategy. The problem is *divided* into smaller problems and solved recursively. The *conquering* phase consists of patching together the answers. Divide-and-conquer is a very powerful use of recursion that we will see many times.

An implementation of mergesort is provided in Figure 7.11. The one-parameter `mergeSort` is just a driver for the four-parameter recursive `mergeSort`.

The `merge` routine is subtle. If a temporary array is declared locally for each recursive call of `merge`, then there could be $\log N$ temporary arrays active at any point. A close examination shows that since `merge` is the last line of `mergeSort`, there only needs to be one

```
1   /**
2    * Mergesort algorithm (driver).
3    */
4   template <typename Comparable>
5   void mergeSort( vector<Comparable> & a )
6   {
7       vector<Comparable> tmpArray( a.size( ) );
8
9       mergeSort( a, tmpArray, 0, a.size( ) - 1 );
10  }
11
12  /**
13   * Internal method that makes recursive calls.
14   * a is an array of Comparable items.
15   * tmpArray is an array to place the merged result.
16   * left is the left-most index of the subarray.
17   * right is the right-most index of the subarray.
18   */
19  template <typename Comparable>
20  void mergeSort( vector<Comparable> & a,
21                  vector<Comparable> & tmpArray, int left, int right )
22  {
23      if( left < right )
24      {
25          int center = ( left + right ) / 2;
26          mergeSort( a, tmpArray, left, center );
27          mergeSort( a, tmpArray, center + 1, right );
28          merge( a, tmpArray, left, center + 1, right );
29      }
30  }
```

Figure 7.11 Mergesort routines

temporary array active at any point, and that the temporary array can be created in the public `mergeSort` driver. Further, we can use any part of the temporary array; we will use the same portion as the input array `a`. This allows the improvement described at the end of this section. Figure 7.12 implements the `merge` routine.

7.6.1 Analysis of Mergesort

Mergesort is a classic example of the techniques used to analyze recursive routines: We have to write a recurrence relation for the running time. We will assume that N is a power of 2 so that we always split into even halves. For $N = 1$, the time to mergesort is constant, which we will denote by 1. Otherwise, the time to mergesort N numbers is equal to the

```
1  /**
2   * Internal method that merges two sorted halves of a subarray.
3   * a is an array of Comparable items.
4   * tmpArray is an array to place the merged result.
5   * leftPos is the left-most index of the subarray.
6   * rightPos is the index of the start of the second half.
7   * rightEnd is the right-most index of the subarray.
8   */
9  template <typename Comparable>
10 void merge( vector<Comparable> & a, vector<Comparable> & tmpArray,
11             int leftPos, int rightPos, int rightEnd )
12 {
13     int leftEnd = rightPos - 1;
14     int tmpPos = leftPos;
15     int numElements = rightEnd - leftPos + 1;
16
17     // Main loop
18     while( leftPos <= leftEnd && rightPos <= rightEnd )
19         if( a[ leftPos ] <= a[ rightPos ] )
20             tmpArray[ tmpPos++ ] = std::move( a[ leftPos++ ] );
21         else
22             tmpArray[ tmpPos++ ] = std::move( a[ rightPos++ ] );
23
24     while( leftPos <= leftEnd )    // Copy rest of first half
25         tmpArray[ tmpPos++ ] = std::move( a[ leftPos++ ] );
26
27     while( rightPos <= rightEnd )  // Copy rest of right half
28         tmpArray[ tmpPos++ ] = std::move( a[ rightPos++ ] );
29
30     // Copy tmpArray back
31     for( int i = 0; i < numElements; ++i, --rightEnd )
32         a[ rightEnd ] = std::move( tmpArray[ rightEnd ] );
33 }
```

Figure 7.12 `merge` routine

time to do two recursive mergesorts of size $N/2$, plus the time to merge, which is linear. The following equations say this exactly:

$$T(1) = 1$$
$$T(N) = 2T(N/2) + N$$

This is a standard recurrence relation, which can be solved several ways. We will show two methods. The first idea is to divide the recurrence relation through by N. The reason for doing this will become apparent soon. This yields

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

This equation is valid for any N that is a power of 2, so we may also write

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

and

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$
$$\vdots$$
$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Now add up all the equations. This means that we add all of the terms on the left-hand side and set the result equal to the sum of all of the terms on the right-hand side. Observe that the term $T(N/2)/(N/2)$ appears on both sides and thus cancels. In fact, virtually all the terms appear on both sides and cancel. This is called **telescoping** a sum. After everything is added, the final result is

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

because all of the other terms cancel and there are $\log N$ equations, and so all the 1s at the end of these equations add up to $\log N$. Multiplying through by N gives the final answer.

$$T(N) = N \log N + N = O(N \log N)$$

Notice that if we did not divide through by N at the start of the solutions, the sum would not telescope. This is why it was necessary to divide through by N.

An alternative method is to substitute the recurrence relation continually on the right-hand side. We have

$$T(N) = 2T(N/2) + N$$

Since we can substitute $N/2$ into the main equation,

$$2T(N/2) = 2(2(T(N/4)) + N/2) = 4T(N/4) + N$$

we have

$$T(N) = 4T(N/4) + 2N$$

Again, by substituting $N/4$ into the main equation, we see that

$$4T(N/4) = 4(2T(N/8) + N/4) = 8T(N/8) + N$$

So we have

$$T(N) = 8T(N/8) + 3N$$

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.

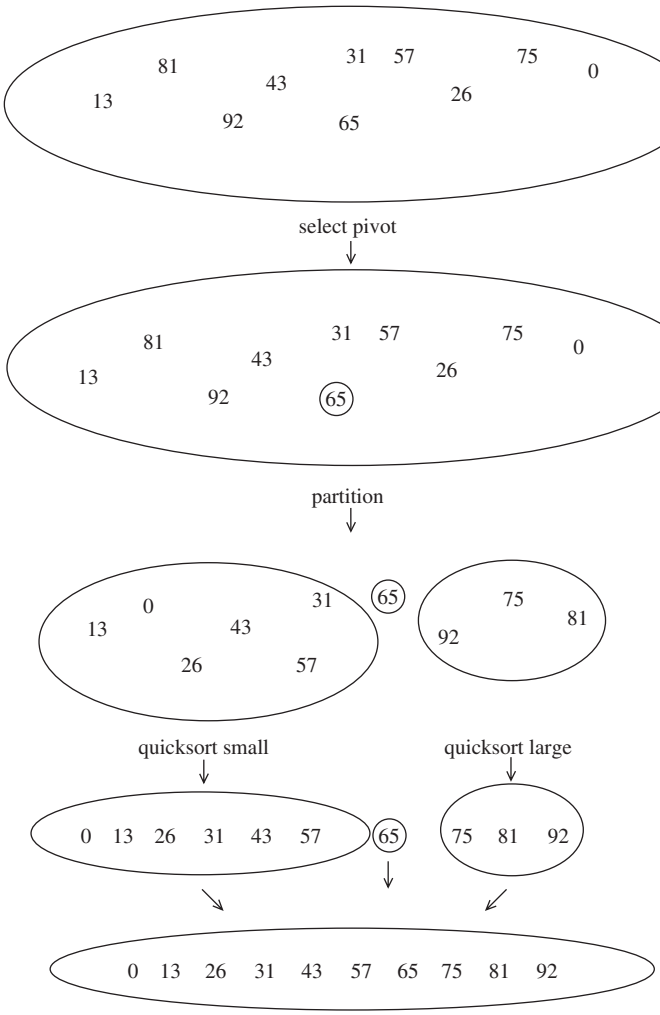


Figure 7.14 The steps of quicksort illustrated by example

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

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

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

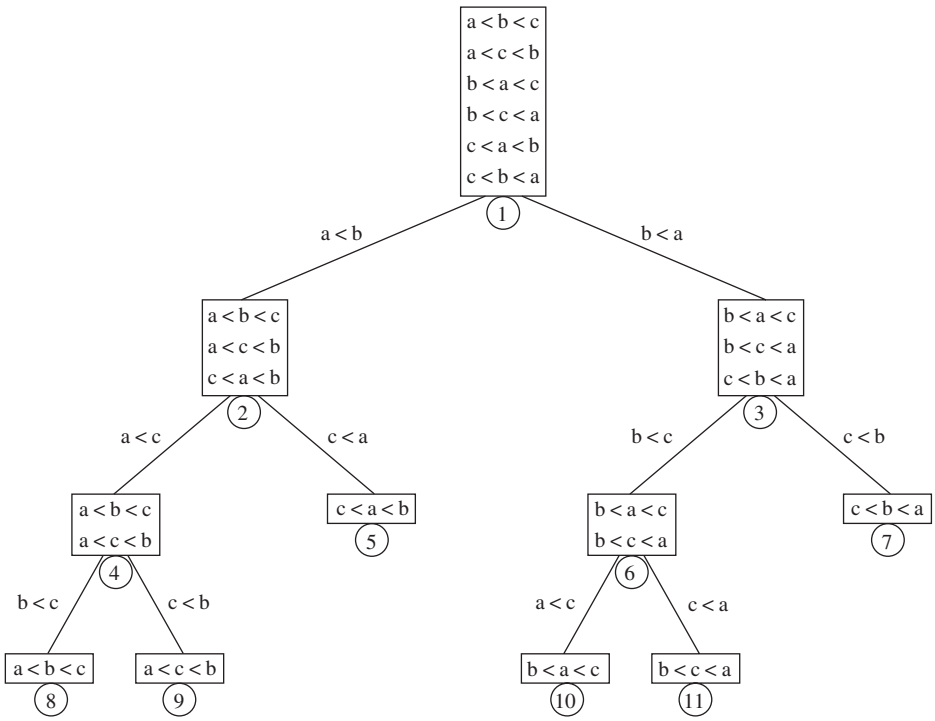


Figure 7.20 A decision tree for three-element sort

leaf. In our case, this algorithm uses three comparisons in the worst case. The average number of comparisons used is equal to the average depth of the leaves. Since a decision tree is large, it follows that there must be some long paths. To prove the lower bounds, all that needs to be shown are some basic tree properties.

Lemma 7.1
Let T be a binary tree of depth d. Then T has at most 2^d leaves.

Proof
The proof is by induction. If $d = 0$, then there is at most one leaf, so the basis is true. Otherwise, we have a root, which cannot be a leaf, and a left and right subtree, each of depth at most $d - 1$. By the induction hypothesis, they can each have at most 2^{d-1} leaves, giving a total of at most 2^d leaves. This proves the lemma.

Lemma 7.2
A binary tree with L leaves must have depth at least $\lceil \log L \rceil$.

Proof
Immediate from the preceding lemma.

Theorem 7.6
Any sorting algorithm that uses only comparisons between elements requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case.

Proof
A decision tree to sort N elements must have $N!$ leaves. The theorem follows from the preceding lemma.

Theorem 7.7
Any sorting algorithm that uses only comparisons between elements requires $\Omega(N \log N)$ comparisons.

Proof
From the previous theorem, $\log(N!)$ comparisons are required.

$$\begin{aligned}
\log(N!) &= \log(N(N-1)(N-2)\cdots(2)(1)) \\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\
&\geq \frac{N}{2} \log \frac{N}{2} \\
&\geq \frac{N}{2} \log N - \frac{N}{2} \\
&= \Omega(N \log N)
\end{aligned}$$

This type of lower-bound argument, when used to prove a worst-case result, is sometimes known as an **information-theoretic** lower bound. The general theorem says that if there are P different possible cases to distinguish, and the questions are of the form YES/NO, then $\lceil \log P \rceil$ questions are always required in some case by any algorithm to solve the problem. It is possible to prove a similar result for the average-case running time of any comparison-based sorting algorithm. This result is implied by the following lemma, which is left as an exercise: Any binary tree with L leaves has an average depth of at least $\log L$.

7.9 Decision-Tree Lower Bounds for Selection Problems

Section 7.8 employed a decision-tree argument to show the fundamental lower bound that any comparison-based sorting algorithm must use roughly $N \log N$ comparisons. In this section, we show additional lower bounds for selection in an N-element collection, specifically

1. $N - 1$ comparisons are necessary to find the smallest item.
2. $N + \lceil \log N \rceil - 2$ comparisons are necessary to find the two smallest items.
3. $\lceil 3N/2 \rceil - O(\log N)$ comparisons are necessary to find the median.

The lower bounds for all these problems, with the exception of finding the median, are tight: Algorithms exist that use exactly the specified number of comparisons. In all our proofs, we assume all items are unique.

Lemma 7.3
If all the leaves in a decision tree are at depth d or higher, the decision tree must have at least 2^d leaves.

Proof
Note that all nonleaf nodes in a decision tree have two children. The proof is by induction and follows Lemma 7.1.

The first lower bound, for finding the smallest item, is the easiest and most trivial to show.

Theorem 7.8
Any comparison-based algorithm to find the smallest element must use at least $N - 1$ comparisons.

Proof
Every element, x, except the smallest element, must be involved in a comparison with some other element, y, in which x is declared larger than y. Otherwise, if there were two different elements that had not been declared larger than any other elements, then either could be the smallest.

Lemma 7.4
The decision tree for finding the smallest of N elements must have at least 2^{N-1} leaves.

Proof
By Theorem 7.8, all leaves in this decision tree are at depth $N - 1$ or higher. Then this lemma follows from Lemma 7.3.

The bound for selection is a little more complicated and requires looking at the structure of the decision tree. It will allow us to prove lower bounds for problems 2 and 3 on our list.

Lemma 7.5
The decision tree for finding the kth smallest of N elements must have at least $\binom{N}{k-1} 2^{N-k}$ leaves.

Proof
Observe that any algorithm that correctly identifies the kth smallest element t must be able to prove that all other elements x are either larger than or smaller than t. Otherwise, it would be giving the same answer regardless of whether x was larger or smaller than t, and the answer cannot be the same in both circumstances. Thus each leaf in the tree, in addition to representing the kth smallest element, also represents the $k - 1$ smallest items that have been identified.

Let T be the decision tree, and consider two sets: $S = \{ x_1, x_2, \ldots, x_{k-1} \}$, representing the $k - 1$ smallest items, and R which are the remaining items, including the

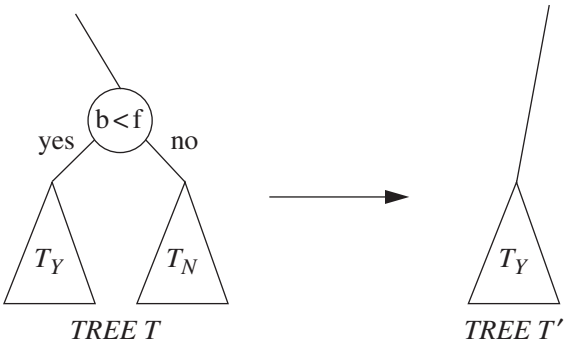


Figure 7.21 Smallest three elements are $S = \{ a, b, c \}$; largest four elements are $R = \{ d, e, f, g \}$; the comparison between b and f for this choice of R and S can be eliminated when forming tree T'

kth smallest. Form a new decision tree, T', by purging any comparisons in T between an element in S and an element in R. Since any element in S is smaller than an element in R, the comparison tree node and its right subtree may be removed from T without any loss of information. Figure 7.21 shows how nodes can be pruned.

Any permutation of R that is fed into T' follows the same path of nodes and leads to the same leaf as a corresponding sequence consisting of a permutation of S followed by the same permutation of R. Since T identifies the overall kth smallest element, and the smallest element in R is that element, it follows that T' identifies the smallest element in R. Thus T' must have at least $2^{|R|-1} = 2^{N-k}$ leaves. These leaves in T' directly correspond to 2^{N-k} leaves representing S. Since there are $\binom{N}{k-1}$ choices for S, there must be at least $\binom{N}{k-1} 2^{N-k}$ leaves in T.

A direct application of Lemma 7.5 allows us to prove the lower bounds for finding the second smallest element and the median.

Theorem 7.9

Any comparison-based algorithm to find the kth smallest element must use at least $N - k + \left\lceil \log \binom{N}{k-1} \right\rceil$ comparisons.

Proof

Immediate from Lemma 7.5 and Lemma 7.2.

Theorem 7.10

Any comparison-based algorithm to find the second smallest element must use at least $N + \lceil \log N \rceil - 2$ comparisons.

Proof

Applying Theorem 7.9, with $k = 2$ yields $N - 2 + \lceil \log N \rceil$.

Theorem 7.11
Any comparison-based algorithm to find the median must use at least $\lceil 3N/2 \rceil - O(\log N)$ comparisons.

Proof
Apply Theorem 7.9, with $k = \lceil N/2 \rceil$.

The lower bound for selection is not tight, nor is it the best known; see the references for details.

7.10 Adversary Lower Bounds

Although decision-tree arguments allowed us to show lower bounds for sorting and some selection problems, generally the bounds that result are not that tight, and sometimes are trivial.

For instance, consider the problem of finding the minimum item. Since there are N possible choices for the minimum, the information theory lower bound that is produced by a decision-tree argument is only $\log N$. In Theorem 7.8, we were able to show the $N - 1$ bound by using what is essentially an **adversary argument**. In this section, we expand on this argument and use it to prove the following lower bound:

4. $\lceil 3N/2 \rceil - 2$ comparisons are necessary to find both the smallest and largest item

Recall our proof that any algorithm to find the smallest item requires at least $N - 1$ comparisons:

Every element, x, except the smallest element, must be involved in a comparison with some other element, y, in which x is declared larger than y. Otherwise, if there were two different elements that had not been declared larger than any other elements, then either could be the smallest.

This is the underlying idea of an adversary argument which has some basic steps:

1. Establish that some basic amount of information must be obtained by any algorithm that solves a problem.
2. In each step of the algorithm, the adversary will maintain an input that is consistent with all the answers that have been provided by the algorithm thus far.
3. Argue that with insufficient steps, there are multiple consistent inputs that would provide different answers to the algorithm; hence, the algorithm has not done enough steps, because if the algorithm were to provide an answer at that point, the adversary would be able to show an input for which the answer is wrong.

To see how this works, we will re-prove the lower bound for finding the smallest element using this proof template.

Theorem 7.8 *(restated)*
Any comparison-based algorithm to find the smallest element must use at least $N - 1$ comparisons.

New Proof

Begin by marking each item as *U* (for unknown). When an item is declared larger than another item, we will change its marking to *E* (for eliminated). This change represents one unit of information. Initially each unknown item has a value of 0, but there have been no comparisons, so this ordering is consistent with prior answers.

A comparison between two items is either between two unknowns or it involves at least one item eliminated from being the minimum. Figure 7.22 shows how our adversary will construct the input values, based on the questioning.

If the comparison is between two unknowns, the first is declared the smaller and the second is automatically eliminated, providing one unit of information. We then assign it (irrevocably) a number larger than 0; the most convenient is the number of eliminated items. If a comparison is between an eliminated number and an unknown, the eliminated number (which is larger than 0 by the prior sentence) will be declared larger, and there will be no changes, no eliminations, and no information obtained. If two eliminated numbers are compared, then they will be different, and a consistent answer can be provided, again with no changes, and no information provided.

At the end, we need to obtain $N-1$ units of information, and each comparison provides only 1 unit at the most; hence, at least $N - 1$ comparisons are necessary.

Lower Bound for Finding the Minimum and Maximum

We can use this same technique to establish a lower bound for finding both the minimum and maximum item. Observe that all but one item must be eliminated from being the smallest, and all but one item must be eliminated from being the largest; thus the total information that any algorithm must acquire is $2N - 2$. However, a comparison $x < y$ eliminates both x from being the maximum and y from being the minimum; thus, a comparison can provide two units of information. Consequently, this argument yields only the trivial $N - 1$ lower bound. Our adversary needs to do more work to ensure that it does not give out two units of information more than it needs to.

To achieve this, each item will initially be unmarked. If it "wins" a comparison (i.e., it is declared larger than some item), it obtains a *W*. If it "loses" a comparison (i.e., it is declared smaller than some item), it obtains an *L*. At the end, all but two items will be *WL*. Our adversary will ensure that it only hands out two units of information if it is comparing two unmarked items. That can happen only $\lfloor N/2 \rfloor$ times; then the remaining information has to be obtained one unit at a time, which will establish the bound.

x	*y*	Answer	Information	New *x*	New *y*
U	*U*	$x < y$	1	No change	Mark *y* as *E* Change *y* to #elim
All others		Consistently	0	No change	No change

Figure 7.22 Adversary constructs input for finding the minimum as algorithm runs

Theorem 7.12

Any comparison-based algorithm to find the minimum and maximum must use at least $\lceil 3N/2 \rceil - 2$ comparisons.

Proof

The basic idea is that if two items are unmarked, the adversary must give out two pieces of information. Otherwise, one of the items has either a W or an L (perhaps both). In that case, with reasonable care, the adversary should be able to avoid giving out two units of information. For instance, if one item, x, has a W and the other item, y, is unmarked, the adversary lets x win again by saying $x > y$. This gives one unit of information for y but no new information for x. It is easy to see that, in principle, there is no reason that the adversary should have to give more than one unit of information out if there is at least one unmarked item involved in the comparison.

It remains to show that the adversary can maintain values that are consistent with its answers. If both items are unmarked, then obviously they can be safely assigned values consistent with the comparison answer; this case yields two units of information.

Otherwise, if one of the items involved in a comparison is unmarked, it can be assigned a value the first time, consistent with the other item in the comparison. This case yields one unit of information.

x	y	Answer	Information	New x	New y
–	–	$x < y$	2	L 0	W 1
L	–	$x < y$	1	L unchanged	W $x + 1$
W *or* WL	–	$x > y$	1	W *or* WL unchanged	L $x - 1$
W *or* WL	W	$x < y$	1 or 0	WL unchanged	W $max(x + 1, y)$
L *or* W *or* WL	L	$x > y$	1 or 0 or 0	WL *or* W *or* WL unchanged	L $min(x - 1, y)$
WL	WL	consistent	0	unchanged	unchanged
– – – L L W	W WL L W WL WL	SYMMETRIC TO AN ABOVE CASE			

Figure 7.23 Adversary constructs input for finding the maximum and minimum as algorithm runs

Otherwise, both items involved in the comparison are marked. If both are *WL*, then we can answer consistently with the current assignment, yielding no information.[1]

Otherwise, at least one of the items has only an *L* or only a *W*. We will allow that item to compare redundantly (if it is an *L* then it loses again; if it is a *W* then it wins again), and its value can be easily adjusted if needed, based on the other item in the comparison (an *L* can be lowered as needed; a *W* can be raised as needed). This yields at most one unit of information for the other item in the comparison, possibly zero. Figure 7.23 summarizes the action of the adversary, making *y* the primary element whose value changes in all cases.

At most $\lfloor N/2 \rfloor$ comparisons yield two units of information, meaning that the remaining information, namely $2N - 2 - 2\lfloor N/2 \rfloor$ units, must each be obtained one comparison at a time. Thus the total number of comparisons that are needed is at least $2N - 2 - \lfloor N/2 \rfloor = \lceil 3N/2 \rceil - 2$.

It is easy to see that this bound is achievable. Pair up the elements, and perform a comparison between each pair. Then find the maximum among the winners and the minimum among the losers.

7.11 Linear-Time Sorts: Bucket Sort and Radix Sort

Although we proved in Section 7.8 that any general sorting algorithm that uses only comparisons requires $\Omega(N \log N)$ time in the worst case, recall that it is still possible to sort in linear time in some special cases.

A simple example is **bucket sort**. For bucket sort to work, extra information must be available. The input $A_1, A_2, \ldots, A_N$ must consist of only positive integers smaller than *M*. (Obviously extensions to this are possible.) If this is the case, then the algorithm is simple: Keep an array called `count`, of size *M*, which is initialized to all 0s. Thus, `count` has *M* cells, or buckets, which are initially empty. When A_i is read, increment `count[`A_i`]` by 1. After all the input is read, scan the `count` array, printing out a representation of the sorted list. This algorithm takes $O(M + N)$; the proof is left as an exercise. If *M* is $O(N)$, then the total is $O(N)$.

Although this algorithm seems to violate the lower bound, it turns out that it does not because it uses a more powerful operation than simple comparisons. By incrementing the appropriate bucket, the algorithm essentially performs an *M*-way comparison in unit time. This is similar to the strategy used in extendible hashing (Section 5.9). This is clearly not in the model for which the lower bound was proven.

This algorithm does, however, question the validity of the model used in proving the lower bound. The model actually is a strong model, because a *general-purpose* sorting algorithm cannot make assumptions about the type of input it can expect to see but must

[1] It is possible that the current assignment for both items has the same number; in such a case we can increase all items whose current value is larger than *y* by 2, and then add 1 to *y* to break the tie.

make decisions based on ordering information only. Naturally, if there is extra information available, we should expect to find a more efficient algorithm, since otherwise the extra information would be wasted.

Although bucket sort seems like much too trivial an algorithm to be useful, it turns out that there are many cases where the input is only small integers, so that using a method like quicksort is really overkill. One such example is **radix sort**.

Radix sort is sometimes known as card sort because it was used until the advent of modern computers to sort old-style punch cards. Suppose we have 10 numbers in the range 0 to 999 that we would like to sort. In general this is N numbers in the range 0 to $b^p - 1$ for some constant p. Obviously we cannot use bucket sort; there would be too many buckets. The trick is to use several passes of bucket sort. The natural algorithm would be to bucket sort by the most significant "digit" (digit is taken to base b), then next most significant, and so on. But a simpler idea is to perform bucket sorts in the reverse order, starting with the least significant "digit" first. Of course, more than one number could fall into the same bucket, and unlike the original bucket sort, these numbers could be different, so we keep them in a list. Each pass is stable: Items that agree in the current digit retain the ordering determined in prior passes. The trace in Figure 7.24 shows the result of sorting 64, 8, 216, 512, 27, 729, 0, 1, 343, 125, which is the first ten cubes arranged randomly (we use 0s to make clear the tens and hundreds digits). After the first pass, the items are sorted on the least significant digit, and in general, after the kth pass, the items are sorted on the k least significant digits. So at the end, the items are completely sorted. To see that the algorithm works, notice that the only possible failure would occur if two numbers came out of the same bucket in the wrong order. But the previous passes ensure that when several numbers enter a bucket, they enter in sorted order according to the k-1 least significant digits. The running time is $O(p(N + b))$ where p is the number of passes, N is the number of elements to sort, and b is the number of buckets.

One application of radix sort is sorting strings. If all the strings have the same length L, then by using buckets for each character, we can implement a radix sort in $O(NL)$ time. The most straightforward way of doing this is shown in Figure 7.25. In our code, we assume that all characters are ASCII, residing in the first 256 positions of the Unicode character set. In each pass, we add an item to the appropriate bucket, and then after all the buckets are populated, we step through the buckets dumping everything back to the array. Notice that when a bucket is populated and emptied in the next pass, the order from the current pass is preserved.

Counting radix sort is an alternative implementation of radix sort that avoids using `vector`s to represent buckets. Instead, we maintain a count of how many items would go in each bucket; this information can go into an array `count`, so that `count[k]` is the number of items that are in bucket `k`. Then we can use another array `offset`, so that `offset[k]`

INITIAL ITEMS:	064, 008, 216, 512, 027, 729, 000, 001, 343, 125
SORTED BY 1's digit:	000, 001, 512, 343, 064, 125, 216, 027, 008, 729
SORTED BY 10's digit:	000, 001, 008, 512, 216, 125, 027, 729, 343, 064
SORTED BY 100's digit:	000, 001, 008, 027, 064, 125, 216, 343, 512, 729

Figure 7.24 Radix sort trace

```
1   /*
2    * Radix sort an array of Strings
3    * Assume all are all ASCII
4    * Assume all have same length
5    */
6   void radixSortA( vector<string> & arr, int stringLen )
7   {
8       const int BUCKETS = 256;
9       vector<vector<string>> buckets( BUCKETS );
10
11      for( int pos = stringLen - 1; pos >= 0; --pos )
12      {
13          for( string & s : arr )
14              buckets[ s[ pos ] ].push_back( std::move( s ) );
15
16          int idx = 0;
17          for( auto & thisBucket : buckets )
18          {
19              for( string & s : thisBucket )
20                  arr[ idx++ ] = std::move( s );
21
22              thisBucket.clear( );
23          }
24      }
25  }
```

Figure 7.25 Simple implementation of radix sort for strings, using an `ArrayList` of buckets

represents the number of items whose value is strictly smaller than `k`. Then when we see a value `k` for the first time in the final scan, `offset[k]` tells us a valid array spot where it can be written to (but we have to use a temporary array for the write), and after that is done, `offset[k]` can be incremented. Counting radix sort thus avoids the need to keep lists. As a further optimization, we can avoid using `offset` by reusing the `count` array. The modification is that we initially have `count[k+1]` represent the number of items that are in bucket `k`. Then after that information is computed, we can scan the `count` array from the smallest to largest index and increment `count[k]` by `count[k-1]`. It is easy to verify that after this scan, the count array stores exactly the same information that would have been stored in `offset`.

Figure 7.26 shows an implementation of counting radix sort. Lines 18 to 27 implement the logic above, assuming that the items are stored in array `in` and the result of a single pass is placed in array `out`. Initially, `in` represents `arr` and `out` represents the temporary array, `buffer`. After each pass, we switch the roles of `in` and `out`. If there are an even number of passes, then at the end, `out` is referencing `arr`, so the sort is complete. Otherwise, we have to move from the `buffer` back into `arr`.

```
1   /*
2    * Counting radix sort an array of Strings
3    * Assume all are all ASCII
4    * Assume all have same length
5    */
6   void countingRadixSort( vectro<string> & arr, int stringLen )
7   {
8       const int BUCKETS = 256;
9
10      int N = arr.size( );
11      vector<string> buffer( N );
12
13      vector<string> *in  = &arr;
14      vector<string> *out = &buffer;
15
16      for( int pos = stringLen - 1; pos >= 0; --pos )
17      {
18          vector<int> count( BUCKETS + 1 );
19
20          for( int i = 0; i < N; ++i )
21              ++count[ (*in)[ i ] [ pos ] + 1 ];
22
23          for( int b = 1; b <= BUCKETS; ++b )
24              count[ b ] += count[ b - 1 ];
25
26          for( int i = 0; i < N; ++i )
27              (*out)[ count[ (*in)[ i ] [ pos ] ]++ ] = std::move( (*in)[ i ] );
28
29            // swap in and out roles
30          std::swap( in, out );
31      }
32
33           // if odd number of passes, in is buffer, out is arr; so move back
34      if( stringLen % 2 == 1 )
35          for( int i = 0; i < arr.size( ); ++i )
36              (*out)[ i ] = std::move( (*in)[ i ] );
37  }
```

Figure 7.26 Counting radix sort for fixed-length strings

Generally, counting radix sort is prefereable to using `vector`s to store buckets, but it can suffer from poor locality (`out` is filled in non-sequentially) and thus, surprisingly, it is not always faster than using a `vector` of `vector`s.

We can extend either version of radix sort to work with variable-length strings. The basic algorithm is to first sort the strings by their length. Instead of looking at all the strings,

```
1   /*
2    * Radix sort an array of Strings
3    * Assume all are all ASCII
4    * Assume all have length bounded by maxLen
5    */
6   void radixSort( vector<string> & arr, int maxLen )
7   {
8       const int BUCKETS = 256;
9
10      vector<vector<string>> wordsByLength( maxLen + 1 );
11      vector<vector<string>> buckets( BUCKETS );
12
13      for( string & s : arr )
14          wordsByLength[ s.length( ) ].push_back( std::move( s ) );
15
16      int idx = 0;
17      for( auto & wordList : wordsByLength )
18          for( string & s : wordList )
19              arr[ idx++ ] = std::move( s );
20
21      int startingIndex = arr.size( );
22      for( int pos = maxLen - 1; pos >= 0; --pos )
23      {
24          startingIndex -= wordsByLength[ pos + 1 ].size( );
25
26          for( int i = startingIndex; i < arr.size( ); ++i )
27              buckets[ arr[ i ][ pos ] ].push_back( std::move( arr[ i ] ) );
28
29          idx = startingIndex;
30          for( auto & thisBucket : buckets )
31          {
32              for( string & s : thisBucket )
33                  arr[ idx++ ] = std::move( s );
34
35              thisBucket.clear( );
36          }
37      }
38  }
```

Figure 7.27 Radix sort for variable-length strings

we can then look only at strings that we know are long enough. Since the string lengths are small numbers, the initial sort by length can be done by—bucket sort! Figure 7.27 shows this implementation of radix sort, with `vector`s to store buckets. Here, the words are grouped into buckets by length at lines 13 and 14 and then placed back into the array at lines 16 to 19. Lines 26 and 27 look at only those strings that have a character at position

pos, by making use of the variable startingIndex, which is maintained at lines 21 and 24. Except for that, lines 21 to 37 in Figure 7.27 are the same as lines 11 to 24 in Figure 7.25.

The running time of this version of radix sort is linear in the total number of characters in all the strings (each character appears exactly once at line 27, and the statement at line 33 executes precisely as many times as the line 27). Radix sort for strings will perform especially well when the characters in the string are drawn from a reasonably small alphabet and when the strings either are relatively short or are very similar. Because the $O(N \log N)$ comparison-based sorting algorithms will generally look only at a small number of characters in each string comparison, once the average string length starts getting large, radix sort's advantage is minimized or evaporates completely.

7.12 External Sorting

So far, all the algorithms we have examined require that the input fit into main memory. There are, however, applications where the input is much too large to fit into memory. This section will discuss **external sorting algorithms**, which are designed to handle very large inputs.

7.12.1 Why We Need New Algorithms

Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable. Shellsort compares elements a[i] and a[i-h_k] in one time unit. Heapsort compares elements a[i] and a[i*2+1] in one time unit. Quicksort, with median-of-three partitioning, requires comparing a[left], a[center], and a[right] in a constant number of time units. If the input is on a tape, then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially. Even if the data are on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.

To see how slow external accesses really are, create a random file that is large, but not too big to fit in main memory. Read the file in and sort it using an efficient algorithm. The time it takes to read the input is certain to be significant compared to the time to sort the input, even though sorting is an $O(N \log N)$ operation and reading the input is only $O(N)$.

7.12.2 Model for External Sorting

The wide variety of mass storage devices makes external sorting much more device dependent than internal sorting. The algorithms that we will consider work on tapes, which are probably the most restrictive storage medium. Since access to an element on tape is done by winding the tape to the correct location, tapes can be efficiently accessed only in sequential order (in either direction).

We will assume that we have at least three tape drives to perform the sorting. We need two drives to do an efficient sort; the third drive simplifies matters. If only one tape drive is present, then we are in trouble: Any algorithm will require $\Omega(N^2)$ tape accesses.

7.12.3 The Simple Algorithm

The basic external sorting algorithm uses the merging algorithm from mergesort. Suppose we have four tapes, T_{a1}, T_{a2}, T_{b1}, T_{b2}, which are two input and two output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes. Suppose the data are initially on T_{a1}. Suppose further that the internal memory can hold (and sort) *M* records at a time. A natural first step is to read *M* records at a time from the input tape, sort the records internally, and then write the sorted records alternately to T_{b1} and T_{b2}. We will call each set of sorted records a **run**. When this is done, we rewind all the tapes. Suppose we have the same input as our example for Shellsort.

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

If $M = 3$, then after the runs are constructed, the tapes will contain the data indicated in the following figure.

T_{a1}							
T_{a2}							
T_{b1}	11	81	94	17	28	99	15
T_{b2}	12	35	96	41	58	75	

Now T_{b1} and T_{b2} contain a group of runs. We take the first run from each tape and merge them, writing the result, which is a run twice as long, onto T_{a1}. Recall that merging two sorted lists is simple; we need almost no memory, since the merge is performed as T_{b1} and T_{b2} advance. Then we take the next run from each tape, merge these, and write the result to T_{a2}. We continue this process, alternating between T_{a1} and T_{a2}, until either T_{b1} or T_{b2} is empty. At this point either both are empty or there is one run left. In the latter case, we copy this run to the appropriate tape. We rewind all four tapes and repeat the same steps, this time using the *a* tapes as input and the *b* tapes as output. This will give runs of $4M$. We continue the process until we get one run of length *N*.

This algorithm will require $\lceil \log(N/M) \rceil$ passes, plus the initial run-constructing pass. For instance, if we have 10 million records of 128 bytes each, and four megabytes of internal memory, then the first pass will create 320 runs. We would then need nine more passes to complete the sort. Our example requires $\lceil \log 13/3 \rceil = 3$ more passes, which are shown in the following figures:

T_{a1}	11	12	35	81	94	96	15
T_{a2}	17	28	41	58	75	99	
T_{b1}							
T_{b2}							

T_{a1}													
T_{a2}													
T_{b1}	11	12	17	28	35	41	58	75	81	94	96	99	
T_{b2}	15												

T_{a1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{a2}													
T_{b1}													
T_{b2}													

7.12.4 Multiway Merge

If we have extra tapes, then we can expect to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a k-way merge.

Merging two runs is done by winding each input tape to the beginning of each run. Then the smaller element is found, placed on an output tape, and the appropriate input tape is advanced. If there are k input tapes, this strategy works the same way, the only difference being that it is slightly more complicated to find the smallest of the k elements. We can find the smallest of these elements by using a priority queue. To obtain the next element to write on the output tape, we perform a `deleteMin` operation. The appropriate input tape is advanced, and if the run on the input tape is not yet completed, we `insert` the new element into the priority queue. Using the same example as before, we distribute the input onto the three tapes.

T_{a1}							
T_{a2}							
T_{a3}							
T_{b1}	11	81	94	41	58	75	
T_{b2}	12	35	96	15			
T_{b3}	17	28	99				

We then need two more passes of three-way merging to complete the sort.

T_{a1}	11	12	17	28	35	81	94	96	99
T_{a2}	15	41	58	75					
T_{a3}									
T_{b1}									
T_{b2}									
T_{b3}									

T_{a1}													
T_{a2}													
T_{a3}													
T_{b1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{b2}													
T_{b3}													

After the initial run construction phase, the number of passes required using k-way merging is $\lceil \log_k(N/M) \rceil$, because the runs get k times as large in each pass. For the example above, the formula is verified, since $\lceil \log_3(13/3) \rceil = 2$. If we have 10 tapes, then $k = 5$, and our large example from the previous section would require $\lceil \log_5 320 \rceil = 4$ passes.

7.12.5 Polyphase Merge

The k-way merging strategy developed in the last section requires the use of $2k$ tapes. This could be prohibitive for some applications. It is possible to get by with only $k + 1$ tapes. As an example, we will show how to perform two-way merging using only three tapes.

Suppose we have three tapes, T_1, T_2, and T_3, and an input file on T_1 that will produce 34 runs. One option is to put 17 runs on each of T_2 and T_3. We could then merge this result onto T_1, obtaining one tape with 17 runs. The problem is that since all the runs are on one tape, we must now put some of these runs on T_2 to perform another merge. The logical way to do this is to copy the first eight runs from T_1 onto T_2 and then perform the merge. This has the effect of adding an extra half pass for every pass we do.

An alternative method is to split the original 34 runs unevenly. Suppose we put 21 runs on T_2 and 13 runs on T_3. We would then merge 13 runs onto T_1 before T_3 was empty. At this point, we could rewind T_1 and T_3, and merge T_1, with 13 runs, and T_2, which has 8 runs, onto T_3. We could then merge 8 runs until T_2 was empty, which would leave 5 runs left on T_1 and 8 runs on T_3. We could then merge T_1 and T_3, and so on. The following table shows the number of runs on each tape after each pass:

	Run Const.	After $T_3 + T_2$	After $T_1 + T_2$	After $T_1 + T_3$	After $T_2 + T_3$	After $T_1 + T_2$	After $T_1 + T_3$	After $T_2 + T_3$
T_1	0	13	5	0	3	1	0	1
T_2	21	8	0	5	2	0	1	0
T_3	13	0	8	3	0	2	1	0

The original distribution of runs makes a great deal of difference. For instance, if 22 runs are placed on T_2, with 12 on T_1, then after the first merge, we obtain 12 runs on T_3 and 10 runs on T_2. After another merge, there are 10 runs on T_1 and 2 runs on T_3. At this point the going gets slow, because we can only merge two sets of runs before T_3 is exhausted. Then T_1 has 8 runs and T_2 has 2 runs. Again, we can only merge two sets of runs, obtaining T_1 with 6 runs and T_3 with 2 runs. After three more passes, T_2 has two

runs and the other tapes are empty. We must copy one run to another tape, and then we can finish the merge.

It turns out that the first distribution we gave is optimal. If the number of runs is a Fibonacci number F_N, then the best way to distribute them is to split them into two Fibonacci numbers F_{N-1} and F_{N-2}. Otherwise, it is necessary to pad the tape with dummy runs in order to get the number of runs up to a Fibonacci number. We leave the details of how to place the initial set of runs on the tapes as an exercise.

We can extend this to a k-way merge, in which case we need kth order Fibonacci numbers for the distribution, where the kth order Fibonacci number is defined as $F^{(k)}(N) = F^{(k)}(N-1) + F^{(k)}(N-2) + \cdots + F^{(k)}(N-k)$, with the appropriate initial conditions $F^{(k)}(N) = 0, 0 \le N \le k-2, F^{(k)}(k-1) = 1$.

7.12.6 Replacement Selection

The last item we will consider is construction of the runs. The strategy we have used so far is the simplest possible: We read as many records as possible and sort them, writing the result to some tape. This seems like the best approach possible, until one realizes that as soon as the first record is written to an output tape, the memory it used becomes available for another record. If the next record on the input tape is larger than the record we have just output, then it can be included in the run.

Using this observation, we can give an algorithm for producing runs. This technique is commonly referred to as **replacement selection**. Initially, M records are read into memory and placed in a priority queue. We perform a `deleteMin`, writing the smallest (valued) record to the output tape. We read the next record from the input tape. If it is larger than the record we have just written, we can add it to the priority queue. Otherwise, it cannot go into the current run. Since the priority queue is smaller by one element, we can store this new element in the dead space of the priority queue until the run is completed and use the element for the next run. Storing an element in the dead space is similar to what is done in heapsort. We continue doing this until the size of the priority queue is zero, at which point the run is over. We start a new run by building a new priority queue, using all the elements in the dead space. Figure 7.28 shows the run construction for the small example we have been using, with $M = 3$. Dead elements are indicated by an asterisk.

In this example, replacement selection produces only three runs, compared with the five runs obtained by sorting. Because of this, a three-way merge finishes in one pass instead of two. If the input is randomly distributed, replacement selection can be shown to produce runs of average length $2M$. For our large example, we would expect 160 runs instead of 320 runs, so a five-way merge would require four passes. In this case, we have not saved a pass, although we might if we get lucky and have 125 runs or less. Since external sorts take so long, every pass saved can make a significant difference in the running time.

As we have seen, it is possible for replacement selection to do no better than the standard algorithm. However, the input is frequently sorted or nearly sorted to start with, in which case replacement selection produces only a few very long runs. This kind of input is common for external sorts and makes replacement selection extremely valuable.

	3 Elements in Heap Array			Output	Next Element Read
	h[1]	h[2]	h[3]		
Run 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	96	17*
	17*	35*	12*	End of Run	Rebuild Heap
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	15*
	58	99	15*	58	End of Tape
	99		15*	99	
			15*	End of Run	Rebuild Heap
Run 3	15			15	

Figure 7.28 Example of run construction

Summary

Sorting is one of the oldest and most well-studied problems in computing. For most general internal sorting applications, an insertion sort, Shellsort, mergesort, or quicksort is the method of choice. The decision regarding which to use depends on the size of the input and on the underlying environment. Insertion sort is appropriate for very small amounts of input. Shellsort is a good choice for sorting moderate amounts of input. With a proper increment sequence, it gives excellent performance and uses only a few lines of code. Mergesort has $O(N \log N)$ worst-case performance but requires additional space. However, the number of comparisons that are used is nearly optimal, because any algorithm that sorts by using only element comparisons must use at least $\lceil \log (N!) \rceil$ comparisons for some input sequence. Quicksort does not by itself provide this worst-case guarantee and is tricky to code. However, it has almost certain $O(N \log N)$ performance and can be combined with heapsort to give an $O(N \log N)$ worst-case guarantee. Strings can be sorted in linear time using radix sort, and this may be a practical alternative to comparison-based sorts in some instances.

Exercises

7.1 Sort the sequence 3, 1, 4, 1, 5, 9, 2, 6, 5 using insertion sort.

7.2 What is the running time of insertion sort if all elements are equal?

7.3 Suppose we exchange elements `a[i]` and `a[i+k]`, which were originally out of order. Prove that at least 1 and at most $2k - 1$ inversions are removed.

7.4 Show the result of running Shellsort on the input $9, 8, 7, 6, 5, 4, 3, 2, 1$ using the increments $\{1, 3, 7\}$.

7.5 a. What is the running time of Shellsort using the two-increment sequence $\{1, 2\}$?
b. Show that for any N, there exists a three-increment sequence such that Shellsort runs in $O(N^{5/3})$ time.
c. Show that for any N, there exists a six-increment sequence such that Shellsort runs in $O(N^{3/2})$ time.

7.6 * a. Prove that the running time of Shellsort is $\Omega(N^2)$ using increments of the form $1, c, c^2, \ldots, c^i$ for any integer c.
** b. Prove that for these increments, the average running time is $\Theta(N^{3/2})$.

* **7.7** Prove that if a k-sorted file is then h-sorted, it remains k-sorted.

** **7.8** Prove that the running time of Shellsort, using the increment sequence suggested by Hibbard, is $\Omega(N^{3/2})$ in the worst case. (*Hint:* You can prove the bound by considering the special case of what Shellsort does when all elements are either 0 or 1.) Set $a[i] = 1$ if i is expressible as a linear combination of $h_t, h_{t-1}, \ldots, h_{\lfloor t/2 \rfloor + 1}$ and 0 otherwise.

7.9 Determine the running time of Shellsort for
a. sorted input
* b. reverse-ordered input

7.10 Do either of the following modifications to the Shellsort routine coded in Figure 7.6 affect the worst-case running time?
a. Before line 8, subtract one from `gap` if it is even.
b. Before line 8, add one to `gap` if it is even.

7.11 Show how heapsort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.

7.12 What is the running time of heapsort for presorted input?

* **7.13** Show that there are inputs that force every `percolateDown` in heapsort to go all the way to a leaf. (*Hint:* Work backward.)

7.14 Rewrite heapsort so that it sorts only items that are in the range `low` to `high`, which are passed as additional parameters.

7.15 Sort $3, 1, 4, 1, 5, 9, 2, 6$ using mergesort.

7.16 How would you implement mergesort without using recursion?

7.17 Determine the running time of mergesort for
a. sorted input
b. reverse-ordered input
c. random input

7.18 In the analysis of mergesort, constants have been disregarded. Prove that the number of comparisons used in the worst case by mergesort is $N\lceil \log N \rceil - 2^{\lceil \log N \rceil} + 1$.

7.19 Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quicksort with median-of-three partitioning and a cutoff of 3.

7.20 Using the quicksort implementation in this chapter, determine the running time of quicksort for
a. sorted input
b. reverse-ordered input
c. random input

7.21 Repeat Exercise 7.20 when the pivot is chosen as
a. the first element
b. the larger of the first two distinct elements
c. a random element
* d. the average of all elements in the set

7.22 a. For the quicksort implementation in this chapter, what is the running time when all keys are equal?
b. Suppose we change the partitioning strategy so that neither `i` nor `j` stops when an element with the same key as the pivot is found. What fixes need to be made in the code to guarantee that quicksort works, and what is the running time when all keys are equal?
c. Suppose we change the partitioning strategy so that `i` stops at an element with the same key as the pivot, but `j` does not stop in a similar case. What fixes need to be made in the code to guarantee that quicksort works, and when all keys are equal, what is the running time of quicksort?

7.23 Suppose we choose the element in the middle position of the array as pivot. Does this make it unlikely that quicksort will require quadratic time?

7.24 Construct a permutation of 20 elements that is as bad as possible for quicksort using median-of-three partitioning and a cutoff of 3.

7.25 The quicksort in the text uses two recursive calls. Remove one of the calls as follows:
a. Rewrite the code so that the second recursive call is unconditionally the last line in quicksort. Do this by reversing the `if/else` and returning after the call to `insertionSort`.
b. Remove the tail recursion by writing a `while` loop and altering `left`.

7.26 Continuing from Exercise 7.25, after part (a),
a. Perform a test so that the smaller subarray is processed by the first recursive call, while the larger subarray is processed by the second recursive call.
b. Remove the tail recursion by writing a `while` loop and altering `left` or `right`, as necessary.
c. Prove that the number of recursive calls is logarithmic in the worst case.

7.27 Suppose the recursive quicksort receives an `int` parameter, `depth`, from the driver that is initially approximately $2 \log N$.
a. Modify the recursive quicksort to call `heapsort` on its current subarray if the level of recursion has reached `depth`. (*Hint:* Decrement `depth` as you make recursive calls; when it is 0, switch to heapsort.)

b. Prove that the worst-case running time of this algorithm is $O(N \log N)$.
c. Conduct experiments to determine how often `heapsort` gets called.
d. Implement this technique in conjunction with tail-recursion removal in Exercise 7.25.
e. Explain why the technique in Exercise 7.26 would no longer be needed.

7.28 When implementing quicksort, if the array contains lots of duplicates, it may be better to perform a three-way partition (into elements less than, equal to, and greater than the pivot) to make smaller recursive calls. Assume three-way comparisons.

a. Give an algorithm that performs a three-way in-place partition of an N-element subarray using only $N - 1$ three-way comparisons. If there are d items equal to the pivot, you may use d additional `Comparable` swaps, above and beyond the two-way partitioning algorithm. (*Hint:* As `i` and `j` move toward each other, maintain five groups of elements as shown below):

```
EQUAL  SMALL  UNKNOWN  LARGE  EQUAL
              i        j
```

b. Prove that using the algorithm above, sorting an N-element array that contains only d different values, takes $O(dN)$ time.

7.29 Write a program to implement the selection algorithm.

7.30 Solve the following recurrence: $T(N) = (1/N)[\sum_{i=0}^{N-1} T(i)] + cN$, $T(0) = 0$.

7.31 A sorting algorithm is **stable** if elements with equal elements are left in the same order as they occur in the input. Which of the sorting algorithms in this chapter are stable and which are not? Why?

7.32 Suppose you are given a sorted list of N elements followed by $f(N)$ randomly ordered elements. How would you sort the entire list if

a. $f(N) = O(1)$?
b. $f(N) = O(\log N)$?
c. $f(N) = O(\sqrt{N})$?
* d. How large can $f(N)$ be for the entire list still to be sortable in $O(N)$ time?

7.33 Prove that any algorithm that finds an element X in a sorted list of N elements requires $\Omega(\log N)$ comparisons.

7.34 Using Stirling's formula, $N! \approx (N/e)^N \sqrt{2\pi N}$, give a precise estimate for $\log(N!)$.

7.35 * a. In how many ways can two sorted arrays of N elements be merged?
* b. Give a nontrivial lower bound on the number of comparisons required to merge two sorted lists of N elements by taking the logarithm of your answer in part (a).

7.36 Prove that merging two sorted arrays of N items requires at least $2N - 1$ comparisons. You must show that if two elements in the merged list are consecutive and from different lists, then they must be compared.

7.37 Consider the following algorithm for sorting six numbers:

- Sort the first three numbers using Algorithm A.
- Sort the second three numbers using Algorithm B.
- Merge the two sorted groups using Algorithm C.

Show that this algorithm is suboptimal, regardless of the choices for Algorithms *A*, *B*, and *C*.

7.38 Write a program that reads N points in a plane and outputs any group of four or more colinear points (i.e., points on the same line). The obvious brute-force algorithm requires $O(N^4)$ time. However, there is a better algorithm that makes use of sorting and runs in $O(N^2 \log N)$ time.

7.39 Show that the two smallest elements among N can be found in $N + \lceil \log N \rceil - 2$ comparisons.

7.40 The following divide-and-conquer algorithm is proposed for finding the simultaneous maximum and minimum: If there is one item, it is the maximum and minimum, and if there are two items, then compare them, and in one comparison you can find the maximum and minimum. Otherwise, split the input into two halves, divided as evenly as possibly (if N is odd, one of the two halves will have one more element than the other). Recursively find the maximum and minimum of each half, and then in two additional comparisons produce the maximum and minimum for the entire problem.

a. Suppose N is a power of 2. What is the exact number of comparisons used by this algorithm?

b. Suppose N is of the form $3 \cdot 2^k$. What is the exact number of comparisons used by this algorithm?

c. Modify the algorithm as follows: When N is even, but not divisible by four, split the input into sizes of $N/2 - 1$ and $N/2 + 1$. What is the exact number of comparisons used by this algorithm?

7.41 Suppose we want to partition N items into G equal-sized groups of size N/G, such that the smallest N/G items are in group 1, the next smallest N/G items are in group 2, and so on. The groups themselves do not have to be sorted. For simplicity, you may assume that N and G are powers of two.

a. Give an $O(N \log G)$ algorithm to solve this problem.

b. Prove an $\Omega(N \log G)$ lower bound to solve this problem using comparison-based algorithms.

*** 7.42** Give a linear-time algorithm to sort N fractions, each of whose numerators and denominators are integers between 1 and N.

7.43 Suppose arrays A and B are both sorted and both contain N elements. Give an $O(\log N)$ algorithm to find the median of $A \cup B$.

7.44 Suppose you have an array of N elements containing only two distinct keys, `true` and `false`. Give an $O(N)$ algorithm to rearrange the list so that all `false` elements precede the `true` elements. You may use only constant extra space.

7.45 Suppose you have an array of N elements, containing three distinct keys, `true`, `false`, and `maybe`. Give an $O(N)$ algorithm to rearrange the list so that all `false` elements precede `maybe` elements, which in turn precede `true` elements. You may use only constant extra space.

7.46 a. Prove that any comparison-based algorithm to sort 4 elements requires 5 comparisons.

b. Give an algorithm to sort 4 elements in 5 comparisons.

7.47 a. Prove that 7 comparisons are required to sort 5 elements using any comparison-based algorithm.

★ b. Give an algorithm to sort 5 elements with 7 comparisons.

7.48 Write an efficient version of Shellsort and compare performance when the following increment sequences are used:

a. Shell's original sequence
b. Hibbard's increments
c. Knuth's increments: $h_i = \frac{1}{2}(3^i + 1)$
d. Gonnet's increments: $h_t = \lfloor \frac{N}{2.2} \rfloor$ and $h_k = \lfloor \frac{h_{k+1}}{2.2} \rfloor$ (with $h_1 = 1$ if $h_2 = 2$)
e. Sedgewick's increments

7.49 Implement an optimized version of quicksort and experiment with combinations of the following:

a. pivot: first element, middle element, random element, median of three, median of five
b. cutoff values from 0 to 20

7.50 Write a routine that reads in two alphabetized files and merges them together, forming a third, alphabetized, file.

7.51 Suppose we implement the median-of-three routine as follows: Find the median of `a[left]`, `a[center]`, `a[right]`, and swap it with `a[right]`. Proceed with the normal partitioning step starting `i` at `left` and `j` at `right-1` (instead of `left+1` and `right-2`).

a. Suppose the input is $2, 3, 4, \ldots, N-1, N, 1$. For this input, what is the running time of this version of quicksort?
b. Suppose the input is in reverse order. For this input, what is the running time of this version of quicksort?

7.52 Prove that any comparison-based sorting algorithm requires $\Omega(N \log N)$ comparisons on average.

7.53 We are given an array that contains N numbers. We want to determine if there are two numbers whose sum equals a given number K. For instance, if the input is 8, 4, 1, 6, and K is 10, then the answer is yes (4 and 6). A number may be used twice. Do the following:

a. Give an $O(N^2)$ algorithm to solve this problem.
b. Give an $O(N \log N)$ algorithm to solve this problem. (*Hint:* Sort the items first. After that is done, you can solve the problem in linear time.)
c. Code both solutions and compare the running times of your algorithms.

7.54 Repeat Exercise 7.53 for four numbers. Try to design an $O(N^2 \log N)$ algorithm. (*Hint:* Compute all possible sums of two elements. Sort these possible sums. Then proceed as in Exercise 7.53.)

7.55 Repeat Exercise 7.53 for three numbers. Try to design an $O(N^2)$ algorithm.

7.56 Consider the following strategy for `percolateDown`: We have a hole at node X. The normal routine is to compare X's children and then move the child up to X if it is larger (in the case of a (*max*)heap) than the element we are trying to place, thereby pushing the hole down; we stop when it is safe to place the new element in the hole. The alternative strategy is to move elements up and the hole down as far as

possible, without testing whether the new cell can be inserted. This would place the new cell in a leaf and probably violate the heap order; to fix the heap order, percolate the new cell up in the normal manner. Write a routine to include this idea, and compare the running time with a standard implementation of heapsort.

7.57 Propose an algorithm to sort a large file using only two tapes.

7.58 a. Show that a lower bound of $N!/2^{2N}$ on the number of heaps is implied by the fact that `buildHeap` uses at most $2N$ comparisons.
b. Use Stirling's formula to expand this bound.

7.59 M is an N-by-N matrix in which the entries in each rows are in increasing order and the entries in each column are in increasing order (reading top to bottom). Consider the problem of determining if x is in M using three-way comparisons (i.e., one comparison of x with $M[i][j]$ tells you either that x is less than, equal to, or greater than $M[i][j]$).
a. Give an algorithm that uses at most $2N - 1$ comparisons.
b. Prove that any algorithm must use at least $2N - 1$ comparisons.

7.60 There is a prize hidden in a box; the value of the prize is a positive integer between 1 and N, and you are given N. To win the prize, you have to guess its value. Your goal is to do it in as few guesses as possible; however, among those guesses, you may only make at most g guesses that are too high. The value g will be specified at the start of the game, and if you make more than g guesses that are too high, you lose. So, for example, if $g = 0$, you then can win in N guesses by simply guessing the sequence $1, 2, 3, \ldots$
a. Suppose $g = \lceil \log N \rceil$. What strategy minimizes the number of guesses?
b. Suppose $g = 1$. Show that you can always win in $O(N^{1/2})$ guesses.
c. Suppose $g = 1$. Show that any algorithm that wins the prize must use $\Omega(N^{1/2})$ guesses.
⋆ d. Give an algorithm and matching lower bound for any constant g.

References

Knuth's book [16] is a comprehensive reference for sorting. Gonnet and Baeza-Yates [5] has some more results, as well as a huge bibliography.

The original paper detailing Shellsort is [29]. The paper by Hibbard [9] suggested the use of the increments $2^k - 1$ and tightened the code by avoiding swaps. Theorem 7.4 is from [19]. Pratt's lower bound, which uses a more complex method than that suggested in the text, can be found in [22]. Improved increment sequences and upper bounds appear in [13], [28], and [31]; matching lower bounds have been shown in [32]. It has been shown that no increment sequence gives an $O(N \log N)$ worst-case running time [20]. The average-case running time for Shellsort is still unresolved. Yao [34] has performed an extremely complex analysis for the three-increment case. The result has yet to be extended to more increments, but has been slightly improved [14]. The paper by Jiang, Li, and Vityani [15] has shown an $\Omega(p N^{1+1/p})$ lower bound on the average-case running time of p-pass Shellsort. Experiments with various increment sequences appear in [30].

Heapsort was invented by Williams [33]; Floyd [4] provided the linear-time algorithm for heap construction. Theorem 7.5 is from [23].

An exact average-case analysis of mergesort has been described in [7]. An algorithm to perform merging in linear time without extra space is described in [12].

Quicksort is from Hoare [10]. This paper analyzes the basic algorithm, describes most of the improvements, and includes the selection algorithm. A detailed analysis and empirical study was the subject of Sedgewick's dissertation [27]. Many of the important results appear in the three papers [24], [25], and [26]. [1] provides a detailed C implementation with some additional improvements, and points out that older implementations of the UNIX *qsort* library routine are easily driven to quadratic behavior. Exercise 7.27 is from [18].

Decision trees and sorting optimality are discussed in Ford and Johnson [5]. This paper also provides an algorithm that almost meets the lower bound in terms of number of comparisons (but not other operations). This algorithm was eventually shown to be slightly suboptimal by Manacher [17].

The selection lower bounds obtained in Theorem 7.9 are from [6]. The lower bound for finding the maximum and minimum simultaneously is from Pohl [21]. The current best lower bound for finding the median is slightly above $2N$ comparisons due to Dor and Zwick [3]; they also have the best upper bound, which is roughly $2.95N$ comparisons [2].

External sorting is covered in detail in [16]. Stable sorting, described in Exercise 7.31, has been addressed by Horvath [11].

1. J. L. Bentley and M. D. McElroy, "Engineering a Sort Function," *Software—Practice and Experience,* 23 (1993), 1249–1265.
2. D. Dor and U. Zwick, "Selecting the Median," *SIAM Journal on Computing*, 28 (1999), 1722–1758.
3. D. Dor and U. Zwick, "Median Selection Requires $(2 + \varepsilon)n$ Comparisons," *SIAM Journal on Discrete Math*, 14 (2001), 312–325.
4. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM,* 7 (1964), 701.
5. L. R. Ford and S. M. Johnson, "A Tournament Problem," *American Mathematics Monthly,* 66 (1959), 387–389.
6. F. Fussenegger and H. Gabow, "A Counting Approach to Lower Bounds for Selection Problems," *Journal of the ACM*, 26 (1979), 227–238.
7. M. Golin and R. Sedgewick, "Exact Analysis of Mergesort," *Fourth SIAM Conference on Discrete Mathematics,* 1988.
8. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures,* 2d ed., Addison-Wesley, Reading, Mass., 1991.
9. T. H. Hibbard, "An Empirical Study of Minimal Storage Sorting," *Communications of the ACM,* 6 (1963), 206–213.
10. C. A. R. Hoare, "Quicksort," *Computer Journal,* 5 (1962), 10–15.
11. E. C. Horvath, "Stable Sorting in Asymptotically Optimal Time and Extra Space," *Journal of the ACM,* 25 (1978), 177–199.
12. B. Huang and M. Langston, "Practical In-place Merging," *Communications of the ACM,* 31 (1988), 348–352.
13. J. Incerpi and R. Sedgewick, "Improved Upper Bounds on Shellsort," *Journal of Computer and System Sciences,* 31 (1985), 210–224.

14. S. Janson and D. E. Knuth, "Shellsort with Three Increments," *Random Structures and Algorithms*, 10 (1997), 125–142.
15. T. Jiang, M. Li, and P. Vitanyi, "A Lower Bound on the Average-Case Complexity of Shellsort," *Journal of the ACM*, 47 (2000), 905–911.
16. D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
17. G. K. Manacher, "The Ford-Johnson Sorting Algorithm Is Not Optimal," *Journal of the ACM,* 26 (1979), 441–456.
18. D. R. Musser, "Introspective Sorting and Selection Algorithms," *Software—Practice and Experience,* 27 (1997), 983–993.
19. A. A. Papernov and G. V. Stasevich, "A Method of Information Sorting in Computer Memories," *Problems of Information Transmission,* 1 (1965), 63–75.
20. C. G. Plaxton, B. Poonen, and T. Suel, "Improved Lower Bounds for Shellsort," *Proceedings of the Thirty-third Annual Symposium on the Foundations of Computer Science* (1992), 226–235.
21. I. Pohl, "A Sorting Problem and Its Complexity," *Communications of the ACM*, 15 (1972), 462–464.
22. V. R. Pratt, *Shellsort and Sorting Networks,* Garland Publishing, New York, 1979. (Originally presented as the author's Ph.D. thesis, Stanford University, 1971.)
23. R. Schaffer and R. Sedgewick, "The Analysis of Heapsort," *Journal of Algorithms,* 14 (1993), 76–100.
24. R. Sedgewick, "Quicksort with Equal Keys," *SIAM Journal on Computing,* 6 (1977), 240–267.
25. R. Sedgewick, "The Analysis of Quicksort Programs," *Acta Informatica,* 7 (1977), 327–355.
26. R. Sedgewick, "Implementing Quicksort Programs," *Communications of the ACM,* 21 (1978), 847–857.
27. R. Sedgewick, *Quicksort,* Garland Publishing, New York, 1978. (Originally presented as the author's Ph.D. thesis, Stanford University, 1975.)
28. R. Sedgewick, "A New Upper Bound for Shellsort," *Journal of Algorithms,* 7 (1986), 159–173.
29. D. L. Shell, "A High-Speed Sorting Procedure," *Communications of the ACM,* 2 (1959), 30–32.
30. M. A. Weiss, "Empirical Results on the Running Time of Shellsort," *Computer Journal,* 34 (1991), 88–91.
31. M. A. Weiss and R. Sedgewick, "More on Shellsort Increment Sequences," *Information Processing Letters,* 34 (1990), 267–270.
32. M. A. Weiss and R. Sedgewick, "Tight Lower Bounds for Shellsort," *Journal of Algorithms,* 11 (1990), 242–251.
33. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM,* 7 (1964), 347–348.
34. A. C. Yao, "An Analysis of $(h, k, 1)$ Shellsort," *Journal of Algorithms,* 1 (1980), 14–50.