7

Internal Sorting

We sort many things in our everyday lives: A handful of cards when playing Bridge; bills and other piles of paper; jars of spices; and so on. And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around. Sorting is also one of the most frequently performed computing tasks. We might sort the records in a database so that we can search the collection efficiently. We might sort the records by zip code so that we can print and mail them more cheaply. We might use sorting as an intrinsic part of an algorithm to solve some other problem, such as when computing the minimum-cost spanning tree (see Section 11.5).

Because sorting is so important, naturally it has been studied intensively and many algorithms have been devised. Some of these algorithms are straightforward adaptations of schemes we use in everyday life. Others are totally alien to how humans do things, having been invented to sort thousands or even millions of records stored on the computer. After years of study, there are still unsolved problems related to sorting. New algorithms are still being developed and refined for special-purpose applications.

While introducing this central problem in computer science, this chapter has a secondary purpose of illustrating issues in algorithm design and analysis. For example, this collection of sorting algorithms shows multiple approaches to using divide-and-conquer. In particular, there are multiple ways to do the dividing: Mergesort divides a list in half; Quicksort divides a list into big values and small values; and Radix Sort divides the problem by working on one digit of the key at a time. Sorting algorithms can also illustrate a wide variety of analysis techniques. We'll find that it is possible for an algorithm to have an average case whose growth rate is significantly smaller than its worse case (Quicksort). We'll see how it is possible to speed up sorting algorithms (both Shellsort and Quicksort) by taking advantage of the best case behavior of another algorithm (Insertion sort). We'll see several examples of how we can tune an algorithm for better performance. We'll see that special case behavior by some algorithms makes them a good solution for

special niche applications (Heapsort). Sorting provides an example of a significant technique for analyzing the lower bound for a problem. Sorting will also be used to motivate the introduction to file processing presented in Chapter 8.

The present chapter covers several standard algorithms appropriate for sorting a collection of records that fit in the computer's main memory. It begins with a discussion of three simple, but relatively slow, algorithms requiring $\Theta(n^2)$ time in the average and worst cases. Several algorithms with considerably better performance are then presented, some with $\Theta(n \log n)$ worst-case running time. The final sorting method presented requires only $\Theta(n)$ worst-case time under special conditions. The chapter concludes with a proof that sorting in general requires $\Omega(n \log n)$ time in the worst case.

7.1 Sorting Terminology and Notation

Except where noted otherwise, input to the sorting algorithms presented in this chapter is a collection of records stored in an array. Records are compared to one another by means of a comparator class, as introduced in Section 4.4. To simplify the discussion we will assume that each record has a key field whose value is extracted from the record by the comparator. The key method of the comparator class is **`prior`**, which returns true when its first argument should appear prior to its second argument in the sorted list. We also assume that for every record type there is a **`swap`** function that can interchange the contents of two records in the array(see the Appendix).

Given a set of records r_1, r_2, ..., r_n with key values k_1, k_2, ..., k_n, the **Sorting Problem** is to arrange the records into any order s such that records r_{s_1}, r_{s_2}, ..., r_{s_n} have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$. In other words, the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

As defined, the Sorting Problem allows input with two or more records that have the same key value. Certain applications require that input not contain duplicate key values. The sorting algorithms presented in this chapter and in Chapter 8 can handle duplicate key values unless noted otherwise.

When duplicate key values are allowed, there might be an implicit ordering to the duplicates, typically based on their order of occurrence within the input. It might be desirable to maintain this initial ordering among duplicates. A sorting algorithm is said to be **stable** if it does not change the relative ordering of records with identical key values. Many, but not all, of the sorting algorithms presented in this chapter are stable, or can be made stable with minor changes.

When comparing two sorting algorithms, the most straightforward approach would seem to be simply program both and measure their running times. An example of such timings is presented in Figure 7.20. However, such a comparison

can be misleading because the running time for many sorting algorithms depends on specifics of the input values. In particular, the number of records, the size of the keys and the records, the allowable range of the key values, and the amount by which the input records are "out of order" can all greatly affect the relative running times for sorting algorithms.

When analyzing sorting algorithms, it is traditional to measure the number of comparisons made between keys. This measure is usually closely related to the running time for the algorithm and has the advantage of being machine and data-type independent. However, in some cases records might be so large that their physical movement might take a significant fraction of the total running time. If so, it might be appropriate to measure the number of swap operations performed by the algorithm. In most applications we can assume that all records and keys are of fixed length, and that a single comparison or a single swap operation requires a constant amount of time regardless of which keys are involved. Some special situations "change the rules" for comparing sorting algorithms. For example, an application with records or keys having widely varying length (such as sorting a sequence of variable length strings) will benefit from a special-purpose sorting technique. Some applications require that a small number of records be sorted, but that the sort be performed frequently. An example would be an application that repeatedly sorts groups of five numbers. In such cases, the constants in the runtime equations that are usually ignored in an asymptotic analysis now become crucial. Finally, some situations require that a sorting algorithm use as little memory as possible. We will note which sorting algorithms require significant extra memory beyond the input array.

7.2 Three $\Theta(n^2)$ Sorting Algorithms

This section presents three simple sorting algorithms. While easy to understand and implement, we will soon see that they are unacceptably slow when there are many records to sort. Nonetheless, there are situations where one of these simple algorithms is the best tool for the job.

7.2.1 Insertion Sort

Imagine that you have a stack of phone bills from the past two years and that you wish to organize them by date. A fairly natural way to do this might be to look at the first two bills and put them in order. Then take the third bill and put it into the right order with respect to the first two, and so on. As you take each bill, you would add it to the sorted pile that you have already made. This naturally intuitive process is the inspiration for our first sorting algorithm, called **Insertion Sort**. Insertion Sort iterates through a list of records. Each record is inserted in turn at the correct position within a sorted list composed of those records already processed. The

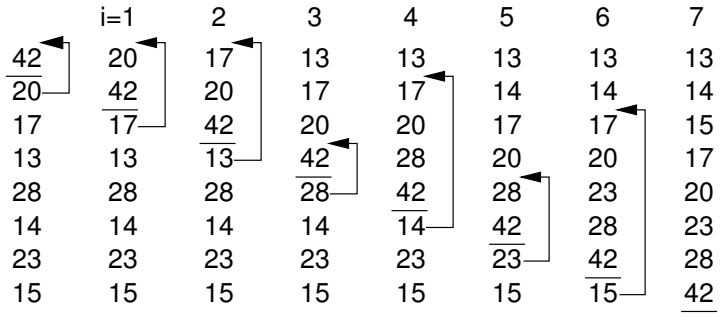

Figure 7.1 An illustration of Insertion Sort. Each column shows the array after the iteration with the indicated value of **i** in the outer **for** loop. Values above the line in each column have been sorted. Arrows indicate the upward motions of records through the array.

following is a C++ implementation. The input is an array of n records stored in array **A**.

```
template <typename E, typename Comp>
void inssort(E A[], int n) { // Insertion Sort
  for (int i=1; i<n; i++)       // Insert i'th record
    for (int j=i; (j>0) && (Comp::prior(A[j], A[j-1])); j--)
      swap(A, j, j-1);
}
```

Consider the case where **inssort** is processing the ith record, which has key value X. The record is moved upward in the array as long as X is less than the key value immediately above it. As soon as a key value less than or equal to X is encountered, **inssort** is done with that record because all records above it in the array must have smaller keys. Figure 7.1 illustrates how Insertion Sort works.

The body of **inssort** is made up of two nested **for** loops. The outer **for** loop is executed $n - 1$ times. The inner **for** loop is harder to analyze because the number of times it executes depends on how many keys in positions 1 to $i - 1$ have a value less than that of the key in position i. In the worst case, each record must make its way to the top of the array. This would occur if the keys are initially arranged from highest to lowest, in the reverse of sorted order. In this case, the number of comparisons will be one the first time through the **for** loop, two the second time, and so on. Thus, the total number of comparisons will be

$$\sum_{i=2}^{n} i \approx n^2/2 = \Theta(n^2).$$

In contrast, consider the best-case cost. This occurs when the keys begin in sorted order from lowest to highest. In this case, every pass through the inner **for** loop will fail immediately, and no values will be moved. The total number

of comparisons will be $n - 1$, which is the number of times the outer **for** loop executes. Thus, the cost for Insertion Sort in the best case is $\Theta(n)$.

While the best case is significantly faster than the worst case, the worst case is usually a more reliable indication of the "typical" running time. However, there are situations where we can expect the input to be in sorted or nearly sorted order. One example is when an already sorted list is slightly disordered by a small number of additions to the list; restoring sorted order using Insertion Sort might be a good idea if we know that the disordering is slight. Examples of algorithms that take advantage of Insertion Sort's near-best-case running time are the Shellsort algorithm of Section 7.3 and the Quicksort algorithm of Section 7.5.

What is the average-case cost of Insertion Sort? When record i is processed, the number of times through the inner **for** loop depends on how far "out of order" the record is. In particular, the inner **for** loop is executed once for each key greater than the key of record i that appears in array positions 0 through $i-1$. For example, in the leftmost column of Figure 7.1 the value 15 is preceded by five values greater than 15. Each such occurrence is called an **inversion**. The number of inversions (i.e., the number of values greater than a given value that occur prior to it in the array) will determine the number of comparisons and swaps that must take place. We need to determine what the average number of inversions will be for the record in position i. We expect on average that half of the keys in the first $i - 1$ array positions will have a value greater than that of the key at position i. Thus, the average case should be about half the cost of the worst case, or around $n^2/4$, which is still $\Theta(n^2)$. So, the average case is no better than the worst case in asymptotic complexity.

Counting comparisons or swaps yields similar results. Each time through the inner **for** loop yields both a comparison and a swap, except the last (i.e., the comparison that fails the inner **for** loop's test), which has no swap. Thus, the number of swaps for the entire sort operation is $n - 1$ less than the number of comparisons. This is 0 in the best case, and $\Theta(n^2)$ in the average and worst cases.

7.2.2 Bubble Sort

Our next sorting algorithm is called **Bubble Sort**. Bubble Sort is often taught to novice programmers in introductory computer science courses. This is unfortunate, because Bubble Sort has no redeeming features whatsoever. It is a relatively slow sort, it is no easier to understand than Insertion Sort, it does not correspond to any intuitive counterpart in "everyday" use, and it has a poor best-case running time. However, Bubble Sort can serve as the inspiration for a better sorting algorithm that will be presented in Section 7.2.3.

Bubble Sort consists of a simple double **for** loop. The first iteration of the inner **for** loop moves through the record array from bottom to top, comparing adjacent keys. If the lower-indexed key's value is greater than its higher-indexed

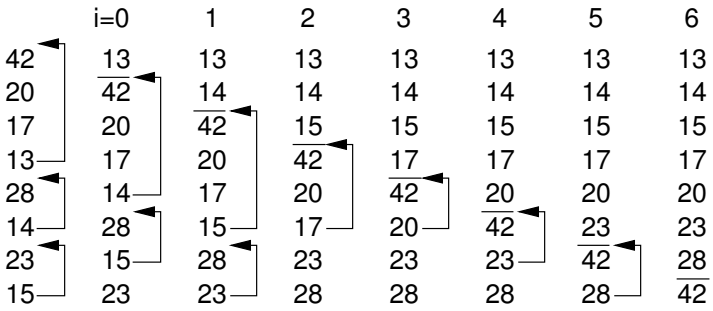

Figure 7.2 An illustration of Bubble Sort. Each column shows the array after the iteration with the indicated value of **i** in the outer **for** loop. Values above the line in each column have been sorted. Arrows indicate the swaps that take place during a given iteration.

neighbor, then the two values are swapped. Once the smallest value is encountered, this process will cause it to "bubble" up to the top of the array. The second pass through the array repeats this process. However, because we know that the smallest value reached the top of the array on the first pass, there is no need to compare the top two elements on the second pass. Likewise, each succeeding pass through the array compares adjacent elements, looking at one less value than the preceding pass. Figure 7.2 illustrates Bubble Sort. A **C**++ implementation is as follows:

```
template <typename E, typename Comp>
void bubsort(E A[], int n) { // Bubble Sort
  for (int i=0; i<n-1; i++)     // Bubble up i'th record
    for (int j=n-1; j>i; j--)
      if (Comp::prior(A[j], A[j-1]))
        swap(A, j, j-1);
}
```

Determining Bubble Sort's number of comparisons is easy. Regardless of the arrangement of the values in the array, the number of comparisons made by the inner **for** loop is always i, leading to a total cost of

$$\sum_{i=1}^{n} i \approx n^2/2 = \Theta(n^2).$$

Bubble Sort's running time is roughly the same in the best, average, and worst cases.

The number of swaps required depends on how often a value is less than the one immediately preceding it in the array. We can expect this to occur for about half the comparisons in the average case, leading to $\Theta(n^2)$ for the expected number of swaps. The actual number of swaps performed by Bubble Sort will be identical to that performed by Insertion Sort.

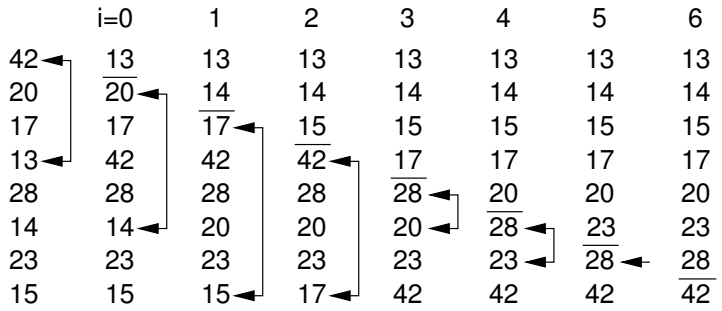

Figure 7.3 An example of Selection Sort. Each column shows the array after the iteration with the indicated value of `i` in the outer `for` loop. Numbers above the line in each column have been sorted and are in their final positions.

7.2.3 Selection Sort

Consider again the problem of sorting a pile of phone bills for the past year. Another intuitive approach might be to look through the pile until you find the bill for January, and pull that out. Then look through the remaining pile until you find the bill for February, and add that behind January. Proceed through the ever-shrinking pile of bills to select the next one in order until you are done. This is the inspiration for our last $\Theta(n^2)$ sort, called **Selection Sort**. The ith pass of Selection Sort "selects" the ith smallest key in the array, placing that record into position i. In other words, Selection Sort first finds the smallest key in an unsorted list, then the second smallest, and so on. Its unique feature is that there are few record swaps. To find the next smallest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record in place. Thus, the total number of swaps required will be $n - 1$ (we get the last record in place "for free").

Figure 7.3 illustrates Selection Sort. Below is a C++ implementation.

```
template <typename E, typename Comp>
void selsort(E A[], int n) { // Selection Sort
  for (int i=0; i<n-1; i++) {   // Select i'th record
    int lowindex = i;           // Remember its index
    for (int j=n-1; j>i; j--)   // Find the least value
      if (Comp::prior(A[j], A[lowindex]))
        lowindex = j;           // Put it in place
    swap(A, i, lowindex);
  }
}
```

Selection Sort (as written here) is essentially a Bubble Sort, except that rather than repeatedly swapping adjacent values to get the next smallest record into place, we instead remember the position of the element to be selected and do one swap at the end. Thus, the number of comparisons is still $\Theta(n^2)$, but the number of swaps is much less than that required by bubble sort. Selection Sort is particularly

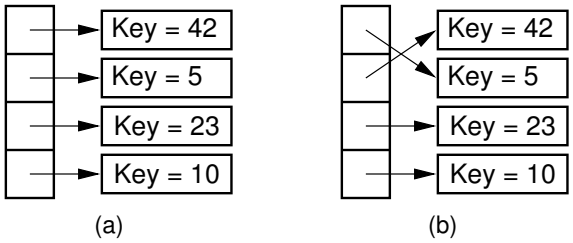

Figure 7.4 An example of swapping pointers to records. (a) A series of four records. The record with key value 42 comes before the record with key value 5. (b) The four records after the top two pointers have been swapped. Now the record with key value 5 comes before the record with key value 42.

advantageous when the cost to do a swap is high, for example, when the elements are long strings or other large records. Selection Sort is more efficient than Bubble Sort (by a constant factor) in most other situations as well.

There is another approach to keeping the cost of swapping records low that can be used by any sorting algorithm even when the records are large. This is to have each element of the array store a pointer to a record rather than store the record itself. In this implementation, a swap operation need only exchange the pointer values; the records themselves do not move. This technique is illustrated by Figure 7.4. Additional space is needed to store the pointers, but the return is a faster swap operation.

7.2.4 The Cost of Exchange Sorting

Figure 7.5 summarizes the cost of Insertion, Bubble, and Selection Sort in terms of their required number of comparisons and swaps[1] in the best, average, and worst cases. The running time for each of these sorts is $\Theta(n^2)$ in the average and worst cases.

The remaining sorting algorithms presented in this chapter are significantly better than these three under typical conditions. But before continuing on, it is instructive to investigate what makes these three sorts so slow. The crucial bottleneck is that only *adjacent* records are compared. Thus, comparisons and moves (in all but Selection Sort) are by single steps. Swapping adjacent records is called an **exchange**. Thus, these sorts are sometimes referred to as **exchange sorts**. The cost of any exchange sort can be at best the total number of steps that the records in the

[1]There is a slight anomaly with Selection Sort. The supposed advantage for Selection Sort is its low number of swaps required, yet Selection Sort's best-case number of swaps is worse than that for Insertion Sort or Bubble Sort. This is because the implementation given for Selection Sort does not avoid a swap in the case where record i is already in position i. One could put in a test to avoid swapping in this situation. But it usually takes more time to do the tests than would be saved by avoiding such swaps.

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Figure 7.5 A comparison of the asymptotic complexities for three simple sorting algorithms.

array must move to reach their "correct" location (i.e., the number of inversions for each record).

What is the average number of inversions? Consider a list $\mathbf{L}$ containing n values. Define $\mathbf{L}_R$ to be $\mathbf{L}$ in reverse. $\mathbf{L}$ has $n(n-1)/2$ distinct pairs of values, each of which could potentially be an inversion. Each such pair must either be an inversion in $\mathbf{L}$ or in $\mathbf{L}_R$. Thus, the total number of inversions in $\mathbf{L}$ and $\mathbf{L}_R$ together is exactly $n(n-1)/2$ for an average of $n(n-1)/4$ per list. We therefore know with certainty that any sorting algorithm which limits comparisons to adjacent items will cost at least $n(n-1)/4 = \Omega(n^2)$ in the average case.

7.3 Shellsort

The next sorting algorithm that we consider is called **Shellsort**, named after its inventor, D.L. Shell. It is also sometimes called the **diminishing increment** sort. Unlike Insertion and Selection Sort, there is no real life intuitive equivalent to Shellsort. Unlike the exchange sorts, Shellsort makes comparisons and swaps between non-adjacent elements. Shellsort also exploits the best-case performance of Insertion Sort. Shellsort's strategy is to make the list "mostly sorted" so that a final Insertion Sort can finish the job. When properly implemented, Shellsort will give substantially better performance than $\Theta(n^2)$ in the worst case.

Shellsort uses a process that forms the basis for many of the sorts presented in the following sections: Break the list into sublists, sort them, then recombine the sublists. Shellsort breaks the array of elements into "virtual" sublists. Each sublist is sorted using an Insertion Sort. Another group of sublists is then chosen and sorted, and so on.

During each iteration, Shellsort breaks the list into disjoint sublists so that each element in a sublist is a fixed number of positions apart. For example, let us assume for convenience that n, the number of values to be sorted, is a power of two. One possible implementation of Shellsort will begin by breaking the list into $n/2$

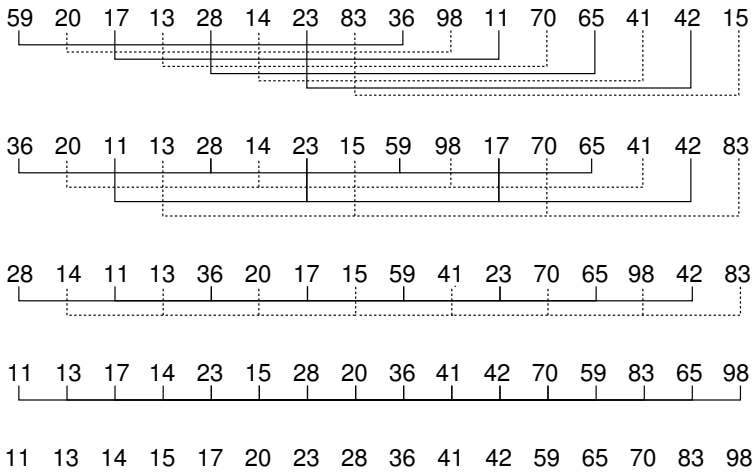

Figure 7.6 An example of Shellsort. Sixteen items are sorted in four passes. The first pass sorts 8 sublists of size 2 and increment 8. The second pass sorts 4 sublists of size 4 and increment 4. The third pass sorts 2 sublists of size 8 and increment 2. The fourth pass sorts 1 list of size 16 and increment 1 (a regular Insertion Sort).

sublists of 2 elements each, where the array index of the 2 elements in each sublist differs by $n/2$. If there are 16 elements in the array indexed from 0 to 15, there would initially be 8 sublists of 2 elements each. The first sublist would be the elements in positions 0 and 8, the second in positions 1 and 9, and so on. Each list of two elements is sorted using Insertion Sort.

The second pass of Shellsort looks at fewer, bigger lists. For our example the second pass would have $n/4$ lists of size 4, with the elements in the list being $n/4$ positions apart. Thus, the second pass would have as its first sublist the 4 elements in positions 0, 4, 8, and 12; the second sublist would have elements in positions 1, 5, 9, and 13; and so on. Each sublist of four elements would also be sorted using an Insertion Sort.

The third pass would be made on two lists, one consisting of the odd positions and the other consisting of the even positions.

The culminating pass in this example would be a "normal" Insertion Sort of all elements. Figure 7.6 illustrates the process for an array of 16 values where the sizes of the increments (the distances between elements on the successive passes) are 8, 4, 2, and 1. Figure 7.7 presents a **C**++ implementation for Shellsort.

Shellsort will work correctly regardless of the size of the increments, *provided that the final pass has increment 1* (i.e., provided the final pass is a regular Insertion Sort). If Shellsort will always conclude with a regular Insertion Sort, then how can it be any improvement on Insertion Sort? The expectation is that each of the (relatively cheap) sublist sorts will make the list "more sorted" than it was before.

```
// Modified version of Insertion Sort for varying increments
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
  for (int i=incr; i<n; i+=incr)
    for (int j=i; (j>=incr) &&
                  (Comp::prior(A[j], A[j-incr])); j-=incr)
      swap(A, j, j-incr);
}

template <typename E, typename Comp>
void shellsort(E A[], int n) { // Shellsort
  for (int i=n/2; i>2; i/=2)      // For each increment
    for (int j=0; j<i; j++)       // Sort each sublist
      inssort2<E,Comp>(&A[j], n-j, i);
  inssort2<E,Comp>(A, n, 1);
}
```

Figure 7.7 An implementation for Shell Sort.

It is not necessarily the case that this will be true, but it is almost always true in practice. When the final Insertion Sort is conducted, the list should be "almost sorted," yielding a relatively cheap final Insertion Sort pass.

Some choices for increments will make Shellsort run more efficiently than others. In particular, the choice of increments described above (2^k, 2^{k-1}, ..., 2, 1) turns out to be relatively inefficient. A better choice is the following series based on division by three: (..., 121, 40, 13, 4, 1).

The analysis of Shellsort is difficult, so we must accept without proof that the average-case performance of Shellsort (for "divisions by three" increments) is $O(n^{1.5})$. Other choices for the increment series can reduce this upper bound somewhat. Thus, Shellsort is substantially better than Insertion Sort, or any of the $\Theta(n^2)$ sorts presented in Section 7.2. In fact, Shellsort is not terrible when compared with the asymptotically better sorts to be presented whenever n is of medium size (thought is tends to be a little slower than these other algorithms when they are well implemented). Shellsort illustrates how we can sometimes exploit the special properties of an algorithm (in this case Insertion Sort) even if in general that algorithm is unacceptably slow.

7.4 Mergesort

A natural approach to problem solving is divide and conquer. In terms of sorting, we might consider breaking the list to be sorted into pieces, process the pieces, and then put them back together somehow. A simple way to do this would be to split the list in half, sort the halves, and then merge the sorted halves together. This is the idea behind **Mergesort**.

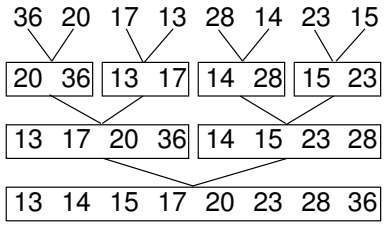

Figure 7.8 An illustration of Mergesort. The first row shows eight numbers that are to be sorted. Mergesort will recursively subdivide the list into sublists of one element each, then recombine the sublists. The second row shows the four sublists of size 2 created by the first merging pass. The third row shows the two sublists of size 4 created by the next merging pass on the sublists of row 2. The last row shows the final sorted list created by merging the two sublists of row 3.

Mergesort is one of the simplest sorting algorithms conceptually, and has good performance both in the asymptotic sense and in empirical running time. Surprisingly, even though it is based on a simple concept, it is relatively difficult to implement in practice. Figure 7.8 illustrates Mergesort. A pseudocode sketch of Mergesort is as follows:

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List L1 = half of the items from inlist;
  List L2 = other half of the items from inlist;
  return merge(mergesort(L1), mergesort(L2));
}
```

Before discussing how to implement Mergesort, we will first examine the merge function. Merging two sorted sublists is quite simple. Function **merge** examines the first element of each sublist and picks the smaller value as the smallest element overall. This smaller value is removed from its sublist and placed into the output list. Merging continues in this way, comparing the front elements of the sublists and continually appending the smaller to the output list until no more input elements remain.

Implementing Mergesort presents a number of technical difficulties. The first decision is how to represent the lists. Mergesort lends itself well to sorting a singly linked list because merging does not require random access to the list elements. Thus, Mergesort is the method of choice when the input is in the form of a linked list. Implementing **merge** for linked lists is straightforward, because we need only remove items from the front of the input lists and append items to the output list. Breaking the input list into two equal halves presents some difficulty. Ideally we would just break the lists into front and back halves. However, even if we know the length of the list in advance, it would still be necessary to traverse halfway down the linked list to reach the beginning of the second half. A simpler method, which does not rely on knowing the length of the list in advance, assigns elements of the

```
template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
  if (left == right) return;        // List of one element
  int mid = (left+right)/2;
  mergesort<E,Comp>(A, temp, left, mid);
  mergesort<E,Comp>(A, temp, mid+1, right);
  for (int i=left; i<=right; i++)   // Copy subarray to temp
    temp[i] = A[i];
  // Do the merge operation back to A
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)      // Left sublist exhausted
      A[curr] = temp[i2++];
    else if (i2 > right)  // Right sublist exhausted
      A[curr] = temp[i1++];
    else if (Comp::prior(temp[i1], temp[i2]))
      A[curr] = temp[i1++];
    else A[curr] = temp[i2++];
  }
}
```

Figure 7.9 Standard implementation for Mergesort.

input list alternating between the two sublists. The first element is assigned to the first sublist, the second element to the second sublist, the third to first sublist, the fourth to the second sublist, and so on. This requires one complete pass through the input list to build the sublists.

When the input to Mergesort is an array, splitting input into two subarrays is easy if we know the array bounds. Merging is also easy if we merge the subarrays into a second array. Note that this approach requires twice the amount of space as any of the sorting methods presented so far, which is a serious disadvantage for Mergesort. It is possible to merge the subarrays without using a second array, but this is extremely difficult to do efficiently and is not really practical. Merging the two subarrays into a second array, while simple to implement, presents another difficulty. The merge process ends with the sorted list in the auxiliary array. Consider how the recursive nature of Mergesort breaks the original array into subarrays, as shown in Figure 7.8. Mergesort is recursively called until subarrays of size 1 have been created, requiring $\log n$ levels of recursion. These subarrays are merged into subarrays of size 2, which are in turn merged into subarrays of size 4, and so on. We need to avoid having each merge operation require a new array. With some difficulty, an algorithm can be devised that alternates between two arrays. A much simpler approach is to copy the sorted sublists to the auxiliary array first, and then merge them back to the original array. Figure 7.9 shows a complete implementation for mergesort following this approach.

An optimized Mergesort implementation is shown in Figure 7.10. It reverses the order of the second subarray during the initial copy. Now the current positions of the two subarrays work inwards from the ends, allowing the end of each subarray

```
template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
  if ((right-left) <= THRESHOLD) { // Small list
    inssort<E,Comp>(&A[left], right-left+1);
    return;
  }
  int i, j, k, mid = (left+right)/2;
  mergesort<E,Comp>(A, temp, left, mid);
  mergesort<E,Comp>(A, temp, mid+1, right);
  // Do the merge operation.  First, copy 2 halves to temp.
  for (i=mid; i>=left; i--) temp[i] = A[i];
  for (j=1; j<=right-mid; j++) temp[right-j+1] = A[j+mid];
  // Merge sublists back to A
  for (i=left,j=right,k=left; k<=right; k++)
    if (Comp::prior(temp[i], temp[j])) A[k] = temp[i++];
    else A[k] = temp[j--];
}
```

Figure 7.10 Optimized implementation for Mergesort.

to act as a sentinel for the other. Unlike the previous implementation, no test is needed to check for when one of the two subarrays becomes empty. This version also uses Insertion Sort to sort small subarrays.

Analysis of Mergesort is straightforward, despite the fact that it is a recursive algorithm. The merging part takes time $\Theta(i)$ where i is the total length of the two subarrays being merged. The array to be sorted is repeatedly split in half until subarrays of size 1 are reached, at which time they are merged to be of size 2, these merged to subarrays of size 4, and so on as shown in Figure 7.8. Thus, the depth of the recursion is $\log n$ for n elements (assume for simplicity that n is a power of two). The first level of recursion can be thought of as working on one array of size n, the next level working on two arrays of size $n/2$, the next on four arrays of size $n/4$, and so on. The bottom of the recursion has n arrays of size 1. Thus, n arrays of size 1 are merged (requiring $\Theta(n)$ total steps), $n/2$ arrays of size 2 (again requiring $\Theta(n)$ total steps), $n/4$ arrays of size 4, and so on. At each of the $\log n$ levels of recursion, $\Theta(n)$ work is done, for a total cost of $\Theta(n \log n)$. This cost is unaffected by the relative order of the values being sorted, thus this analysis holds for the best, average, and worst cases.

7.5 Quicksort

While Mergesort uses the most obvious form of divide and conquer (split the list in half then sort the halves), it is not the only way that we can break down the sorting problem. And we saw that doing the merge step for Mergesort when using an array implementation is not so easy. So perhaps a different divide and conquer strategy might turn out to be more efficient?

Quicksort is aptly named because, when properly implemented, it is the fastest known general-purpose in-memory sorting algorithm in the average case. It does not require the extra array needed by Mergesort, so it is space efficient as well. Quicksort is widely used, and is typically the algorithm implemented in a library sort routine such as the UNIX **`qsort`** function. Interestingly, Quicksort is hampered by exceedingly poor worst-case performance, thus making it inappropriate for certain applications.

Before we get to Quicksort, consider for a moment the practicality of using a Binary Search Tree for sorting. You could insert all of the values to be sorted into the BST one by one, then traverse the completed tree using an inorder traversal. The output would form a sorted list. This approach has a number of drawbacks, including the extra space required by BST pointers and the amount of time required to insert nodes into the tree. However, this method introduces some interesting ideas. First, the root of the BST (i.e., the first node inserted) splits the list into two sublists: The left subtree contains those values in the list less than the root value while the right subtree contains those values in the list greater than or equal to the root value. Thus, the BST implicitly implements a "divide and conquer" approach to sorting the left and right subtrees. Quicksort implements this concept in a much more efficient way.

Quicksort first selects a value called the **pivot**. (This is conceptually like the root node's value in the BST.) Assume that the input array contains k values less than the pivot. The records are then rearranged in such a way that the k values less than the pivot are placed in the first, or leftmost, k positions in the array, and the values greater than or equal to the pivot are placed in the last, or rightmost, $n-k$ positions. This is called a **partition** of the array. The values placed in a given partition need not (and typically will not) be sorted with respect to each other. All that is required is that all values end up in the correct partition. The pivot value itself is placed in position k. Quicksort then proceeds to sort the resulting subarrays now on either side of the pivot, one of size k and the other of size $n-k-1$. How are these values sorted? Because Quicksort is such a good algorithm, using Quicksort on the subarrays would be appropriate.

Unlike some of the sorts that we have seen earlier in this chapter, Quicksort might not seem very "natural" in that it is not an approach that a person is likely to use to sort real objects. But it should not be too surprising that a really efficient sort for huge numbers of abstract objects on a computer would be rather different from our experiences with sorting a relatively few physical objects.

The **C**++ code for Quicksort is shown in Figure 7.11. Parameters **`i`** and **`j`** define the left and right indices, respectively, for the subarray being sorted. The initial call to Quicksort would be **`qsort(array, 0, n-1)`**.

Function **`partition`** will move records to the appropriate partition and then return **`k`**, the first position in the right partition. Note that the pivot value is initially

```
template <typename E, typename Comp>
void qsort(E A[], int i, int j) { // Quicksort
  if (j <= i) return; // Don't sort 0 or 1 element
  int pivotindex = findpivot(A, i, j);
  swap(A, pivotindex, j);    // Put pivot at end
  // k will be the first position in the right subarray
  int k = partition<E,Comp>(A, i-1, j, A[j]);
  swap(A, k, j);             // Put pivot in place
  qsort<E,Comp>(A, i, k-1);
  qsort<E,Comp>(A, k+1, j);
}
```

Figure 7.11 Implementation for Quicksort.

placed at the end of the array (position **j**). Thus, **partition** must not affect the value of array position **j**. After partitioning, the pivot value is placed in position **k**, which is its correct position in the final, sorted array. By doing so, we guarantee that at least one value (the pivot) will not be processed in the recursive calls to **qsort**. Even if a bad pivot is selected, yielding a completely empty partition to one side of the pivot, the larger partition will contain at most $n - 1$ elements.

Selecting a pivot can be done in many ways. The simplest is to use the first key. However, if the input is sorted or reverse sorted, this will produce a poor partitioning with all values to one side of the pivot. It is better to pick a value at random, thereby reducing the chance of a bad input order affecting the sort. Unfortunately, using a random number generator is relatively expensive, and we can do nearly as well by selecting the middle position in the array. Here is a simple **findpivot** function:

```
template <typename E>
inline int findpivot(E A[], int i, int j)
  { return (i+j)/2; }
```

We now turn to function **partition**. If we knew in advance how many keys are less than the pivot, **partition** could simply copy elements with key values less than the pivot to the low end of the array, and elements with larger keys to the high end. Because we do not know in advance how many keys are less than the pivot, we use a clever algorithm that moves indices inwards from the ends of the subarray, swapping values as necessary until the two indices meet. Figure 7.12 shows a C++ implementation for the partition step.

Figure 7.13 illustrates **partition**. Initially, variables **l** and **r** are immediately outside the actual bounds of the subarray being partitioned. Each pass through the outer **do** loop moves the counters **l** and **r** inwards, until eventually they meet. Note that at each iteration of the inner **while** loops, the bounds are moved prior to checking against the pivot value. This ensures that progress is made by each **while** loop, even when the two values swapped on the last iteration of the **do** loop were equal to the pivot. Also note the check that **r > l** in the second **while**

```
template <typename E, typename Comp>
inline int partition(E A[], int l, int r, E& pivot) {
  do {             // Move the bounds inward until they meet
    while (Comp::prior(A[++l], pivot));  // Move l right and
    while ((l < r) && Comp::prior(pivot, A[--r])); // r left
    swap(A, l, r);               // Swap out-of-place values
  } while (l < r);               // Stop when they cross
  return l;      // Return first position in right partition
}
```

Figure 7.12 The Quicksort partition implementation.

Initial	72	6	57	88	85	42	83	73	48	60
	l									r
Pass 1	72	6	57	88	85	42	83	73	48	60
	l								r	
Swap 1	48	6	57	88	85	42	83	73	72	60
	l								r	
Pass 2	48	6	57	88	85	42	83	73	72	60
				l		r				
Swap 2	48	6	57	42	85	88	83	73	72	60
				l		r				
Pass 3	48	6	57	42	85	88	83	73	72	60
					l,r					

Figure 7.13 The Quicksort partition step. The first row shows the initial positions for a collection of ten key values. The pivot value is 60, which has been swapped to the end of the array. The **do** loop makes three iterations, each time moving counters **l** and **r** inwards until they meet in the third pass. In the end, the left partition contains four values and the right partition contains six values. Function **qsort** will place the pivot value into position 4.

loop. This ensures that **r** does not run off the low end of the partition in the case where the pivot is the least value in that partition. Function **partition** returns the first index of the right partition so that the subarray bound for the recursive calls to **qsort** can be determined. Figure 7.14 illustrates the complete Quicksort algorithm.

To analyze Quicksort, we first analyze the **findpivot** and **partition** functions operating on a subarray of length k. Clearly, **findpivot** takes constant time. Function **partition** contains a **do** loop with two nested **while** loops. The total cost of the partition operation is constrained by how far **l** and **r** can move inwards. In particular, these two bounds variables together can move a

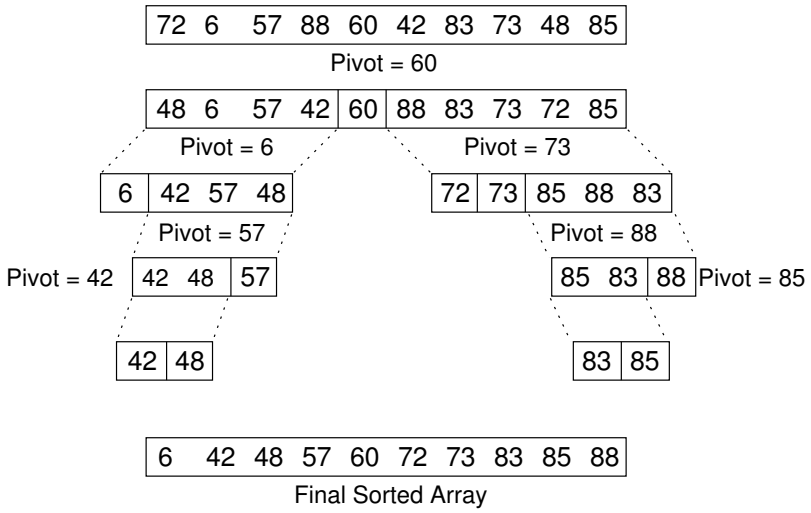

Figure 7.14 An illustration of Quicksort.

total of s steps for a subarray of length s. However, this does not directly tell us how much work is done by the nested **while** loops. The **do** loop as a whole is guaranteed to move both **l** and **r** inward at least one position on each first pass. Each **while** loop moves its variable at least once (except in the special case where **r** is at the left edge of the array, but this can happen only once). Thus, we see that the **do** loop can be executed at most s times, the total amount of work done moving **l** and **r** is s, and each **while** loop can fail its test at most s times. The total work for the entire **partition** function is therefore $\Theta(s)$.

Knowing the cost of **findpivot** and **partition**, we can determine the cost of Quicksort. We begin with a worst-case analysis. The worst case will occur when the pivot does a poor job of breaking the array, that is, when there are no elements in one partition, and $n - 1$ elements in the other. In this case, the divide and conquer strategy has done a poor job of dividing, so the conquer phase will work on a subproblem only one less than the size of the original problem. If this happens at each partition step, then the total cost of the algorithm will be

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

In the worst case, Quicksort is $\Theta(n^2)$. This is terrible, no better than Bubble Sort.[2] When will this worst case occur? Only when each pivot yields a bad partitioning of the array. If the pivot values are selected at random, then this is extremely unlikely to happen. When selecting the middle position of the current subarray, it

[2]The worst insult that I can think of for a sorting algorithm.

is still unlikely to happen. It does not take many good partitionings for Quicksort to work fairly well.

Quicksort's best case occurs when **`findpivot`** always breaks the array into two equal halves. Quicksort repeatedly splits the array into smaller partitions, as shown in Figure 7.14. In the best case, the result will be $\log n$ levels of partitions, with the top level having one array of size n, the second level two arrays of size $n/2$, the next with four arrays of size $n/4$, and so on. Thus, at each level, all partition steps for that level do a total of n work, for an overall cost of $n \log n$ work when Quicksort finds perfect pivots.

Quicksort's average-case behavior falls somewhere between the extremes of worst and best case. Average-case analysis considers the cost for all possible arrangements of input, summing the costs and dividing by the number of cases. We make one reasonable simplifying assumption: At each partition step, the pivot is equally likely to end in any position in the (sorted) array. In other words, the pivot is equally likely to break an array into partitions of sizes 0 and $n-1$, or 1 and $n-2$, and so on.

Given this assumption, the average-case cost is computed from the following equation:

$$\mathbf{T}(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}[\mathbf{T}(k) + \mathbf{T}(n-1-k)], \quad \mathbf{T}(0) = \mathbf{T}(1) = c.$$

This equation is in the form of a recurrence relation. Recurrence relations are discussed in Chapters 2 and 14, and this one is solved in Section 14.2.4. This equation says that there is one chance in n that the pivot breaks the array into subarrays of size 0 and $n-1$, one chance in n that the pivot breaks the array into subarrays of size 1 and $n-2$, and so on. The expression "$\mathbf{T}(k) + \mathbf{T}(n-1-k)$" is the cost for the two recursive calls to Quicksort on two arrays of size k and $n-1-k$. The initial cn term is the cost of doing the **`findpivot`** and **`partition`** steps, for some constant c. The closed-form solution to this recurrence relation is $\Theta(n \log n)$. Thus, Quicksort has average-case cost $\Theta(n \log n)$.

This is an unusual situation that the average case cost and the worst case cost have asymptotically different growth rates. Consider what "average case" actually means. We compute an average cost for inputs of size n by summing up for every possible input of size n the product of the running time cost of that input times the probability that that input will occur. To simplify things, we assumed that every permutation is equally likely to occur. Thus, finding the average means summing up the cost for every permutation and dividing by the number of inputs ($n!$). We know that some of these $n!$ inputs cost $\mathrm{O}(n^2)$. But the sum of all the permutation costs has to be $(n!)(\mathrm{O}(n \log n))$. Given the extremely high cost of the worst inputs, there must be very few of them. In fact, there cannot be a constant fraction of the inputs with cost $\mathrm{O}(n^2)$. Even, say, 1% of the inputs with cost $\mathrm{O}(n^2)$ would lead to

an average cost of $\mathrm{O}(n^2)$. Thus, as n grows, the fraction of inputs with high cost must be going toward a limit of zero. We can conclude that Quicksort will have good behavior if we can avoid those very few bad input permutations.

The running time for Quicksort can be improved (by a constant factor), and much study has gone into optimizing this algorithm. The most obvious place for improvement is the **`findpivot`** function. Quicksort's worst case arises when the pivot does a poor job of splitting the array into equal size subarrays. If we are willing to do more work searching for a better pivot, the effects of a bad pivot can be decreased or even eliminated. One good choice is to use the "median of three" algorithm, which uses as a pivot the middle of three randomly selected values. Using a random number generator to choose the positions is relatively expensive, so a common compromise is to look at the first, middle, and last positions of the current subarray. However, our simple **`findpivot`** function that takes the middle value as its pivot has the virtue of making it highly unlikely to get a bad input by chance, and it is quite cheap to implement. This is in sharp contrast to selecting the first or last element as the pivot, which would yield bad performance for many permutations that are nearly sorted or nearly reverse sorted.

A significant improvement can be gained by recognizing that Quicksort is relatively slow when n is small. This might not seem to be relevant if most of the time we sort large arrays, nor should it matter how long Quicksort takes in the rare instance when a small array is sorted because it will be fast anyway. But you should notice that Quicksort itself sorts many, many small arrays! This happens as a natural by-product of the divide and conquer approach.

A simple improvement might then be to replace Quicksort with a faster sort for small numbers, say Insertion Sort or Selection Sort. However, there is an even better — and still simpler — optimization. When Quicksort partitions are below a certain size, do nothing! The values within that partition will be out of order. However, we do know that all values in the array to the left of the partition are smaller than all values in the partition. All values in the array to the right of the partition are greater than all values in the partition. Thus, even if Quicksort only gets the values to "nearly" the right locations, the array will be close to sorted. This is an ideal situation in which to take advantage of the best-case performance of Insertion Sort. The final step is a single call to Insertion Sort to process the entire array, putting the elements into final sorted order. Empirical testing shows that the subarrays should be left unordered whenever they get down to nine or fewer elements.

The last speedup to be considered reduces the cost of making recursive calls. Quicksort is inherently recursive, because each Quicksort operation must sort two sublists. Thus, there is no simple way to turn Quicksort into an iterative algorithm. However, Quicksort can be implemented using a stack to imitate recursion, as the amount of information that must be stored is small. We need not store copies of a

subarray, only the subarray bounds. Furthermore, the stack depth can be kept small if care is taken on the order in which Quicksort's recursive calls are executed. We can also place the code for **`findpivot`** and **`partition`** inline to eliminate the remaining function calls. Note however that by not processing sublists of size nine or less as suggested above, about three quarters of the function calls will already have been eliminated. Thus, eliminating the remaining function calls will yield only a modest speedup.

7.6 Heapsort

Our discussion of Quicksort began by considering the practicality of using a binary search tree for sorting. The BST requires more space than the other sorting methods and will be slower than Quicksort or Mergesort due to the relative expense of inserting values into the tree. There is also the possibility that the BST might be unbalanced, leading to a $\Theta(n^2)$ worst-case running time. Subtree balance in the BST is closely related to Quicksort's partition step. Quicksort's pivot serves roughly the same purpose as the BST root value in that the left partition (subtree) stores values less than the pivot (root) value, while the right partition (subtree) stores values greater than or equal to the pivot (root).

A good sorting algorithm can be devised based on a tree structure more suited to the purpose. In particular, we would like the tree to be balanced, space efficient, and fast. The algorithm should take advantage of the fact that sorting is a special-purpose application in that all of the values to be stored are available at the start. This means that we do not necessarily need to insert one value at a time into the tree structure.

Heapsort is based on the heap data structure presented in Section 5.5. Heapsort has all of the advantages just listed. The complete binary tree is balanced, its array representation is space efficient, and we can load all values into the tree at once, taking advantage of the efficient **`buildheap`** function. The asymptotic performance of Heapsort is $\Theta(n \log n)$ in the best, average, and worst cases. It is not as fast as Quicksort in the average case (by a constant factor), but Heapsort has special properties that will make it particularly useful when sorting data sets too large to fit in main memory, as discussed in Chapter 8.

A sorting algorithm based on max-heaps is quite straightforward. First we use the heap building algorithm of Section 5.5 to convert the array into max-heap order. Then we repeatedly remove the maximum value from the heap, restoring the heap property each time that we do so, until the heap is empty. Note that each time we remove the maximum element from the heap, it is placed at the end of the array. Assume the n elements are stored in array positions 0 through $n-1$. After removing the maximum value from the heap and readjusting, the maximum value will now be placed in position $n-1$ of the array. The heap is now considered to be

of size $n-1$. Removing the new maximum (root) value places the second largest value in position $n-2$ of the array. After removing each of the remaining values in turn, the array will be properly sorted from least to greatest. This is why Heapsort uses a max-heap rather than a min-heap as might have been expected. Figure 7.15 illustrates Heapsort. The complete **C**++ implementation is as follows:

```
template <typename E, typename Comp>
void heapsort(E A[], int n) { // Heapsort
  E maxval;
  heap<E,Comp> H(A, n, n);     // Build the heap
  for (int i=0; i<n; i++)         // Now sort
    maxval = H.removefirst();     // Place maxval at end
}
```

Because building the heap takes $\Theta(n)$ time (see Section 5.5), and because n deletions of the maximum element each take $\Theta(\log n)$ time, we see that the entire Heapsort operation takes $\Theta(n \log n)$ time in the worst, average, and best cases. While typically slower than Quicksort by a constant factor, Heapsort has one special advantage over the other sorts studied so far. Building the heap is relatively cheap, requiring $\Theta(n)$ time. Removing the maximum element from the heap requires $\Theta(\log n)$ time. Thus, if we wish to find the k largest elements in an array, we can do so in time $\Theta(n + k \log n)$. If k is small, this is a substantial improvement over the time required to find the k largest elements using one of the other sorting methods described earlier (many of which would require sorting all of the array first). One situation where we are able to take advantage of this concept is in the implementation of Kruskal's minimum-cost spanning tree (MST) algorithm of Section 11.5.2. That algorithm requires that edges be visited in ascending order (so, use a min-heap), but this process stops as soon as the MST is complete. Thus, only a relatively small fraction of the edges need be sorted.

7.7 Binsort and Radix Sort

Imagine that for the past year, as you paid your various bills, you then simply piled all the paperwork onto the top of a table somewhere. Now the year has ended and its time to sort all of these papers by what the bill was for (phone, electricity, rent, etc.) and date. A pretty natural approach is to make some space on the floor, and as you go through the pile of papers, put the phone bills into one pile, the electric bills into another pile, and so on. Once this initial assignment of bills to piles is done (in one pass), you can sort each pile by date relatively quickly because they are each fairly small. This is the basic idea behind a Binsort.

Section 3.9 presented the following code fragment to sort a permutation of the numbers 0 through $n-1$:

```
  for (i=0; i<n; i++)
     B[A[i]] = A[i];
```

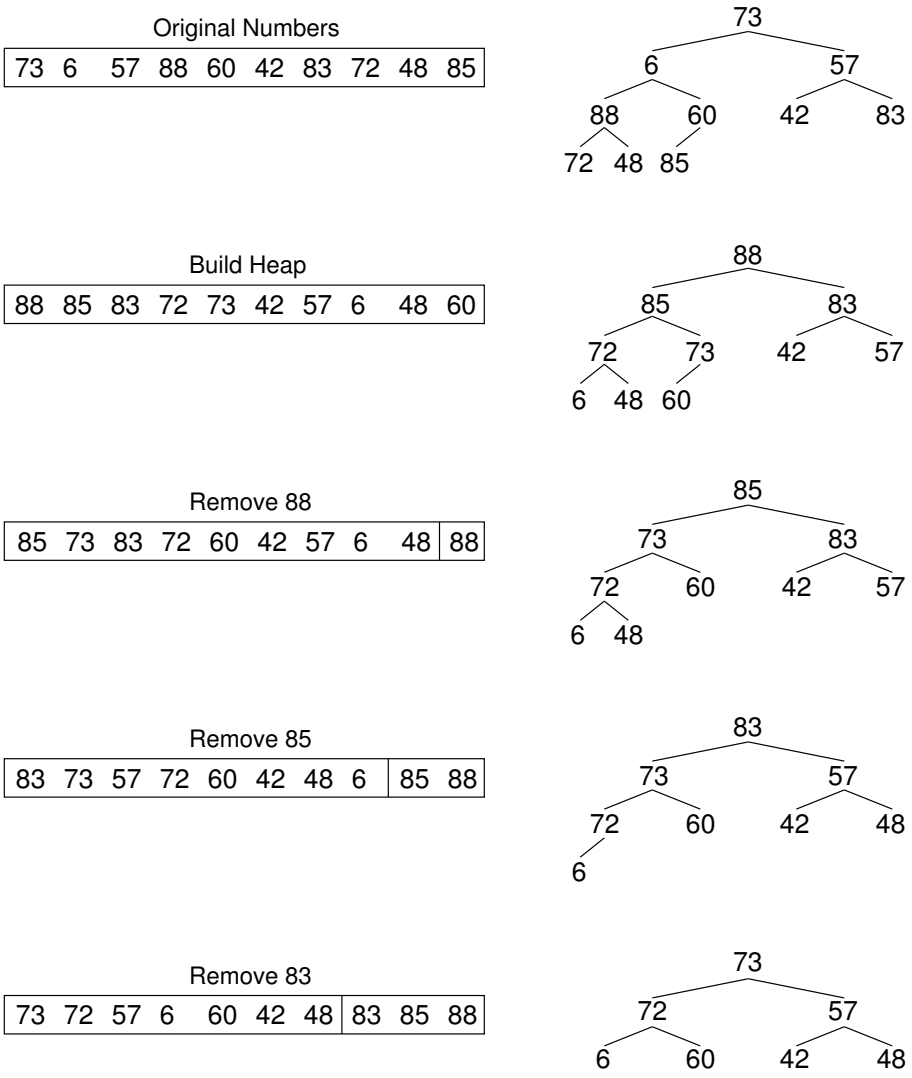

Figure 7.15 An illustration of Heapsort. The top row shows the values in their original order. The second row shows the values after building the heap. The third row shows the result of the first **`removefirst`** operation on key value 88. Note that 88 is now at the end of the array. The fourth row shows the result of the second **`removefirst`** operation on key value 85. The fifth row shows the result of the third **`removefirst`** operation on key value 83. At this point, the last three positions of the array hold the three greatest values in sorted order. Heapsort continues in this manner until the entire array is sorted.

```
template <typename E, class getKey>
void binsort(E A[], int n) {
  List<E> B[MaxKeyValue];
  E item;
  for (int i=0; i<n; i++) B[A[i]].append(getKey::key(A[i]));
  for (int i=0; i<MaxKeyValue; i++)
    for (B[i].setStart(); B[i].getValue(item); B[i].next())
      output(item);
}
```

Figure 7.16 The extended Binsort algorithm.

Here the key value is used to determine the position for a record in the final sorted array. This is the most basic example of a **Binsort**, where key values are used to assign records to **bins**. This algorithm is extremely efficient, taking $\Theta(n)$ time regardless of the initial ordering of the keys. This is far better than the performance of any sorting algorithm that we have seen so far. The only problem is that this algorithm has limited use because it works only for a permutation of the numbers from 0 to $n-1$.

We can extend this simple Binsort algorithm to be more useful. Because Binsort must perform direct computation on the key value (as opposed to just asking which of two records comes first as our previous sorting algorithms did), we will assume that the records use an integer key type. We further assume that it can be extracted from a record using the **key** method supplied by a template parameter class named **getKey**.

The simplest extension is to allow for duplicate values among the keys. This can be done by turning array slots into arbitrary-length bins by turning **B** into an array of linked lists. In this way, all records with key value i can be placed in bin **B[i]**. A second extension allows for a key range greater than n. For example, a set of n records might have keys in the range 1 to $2n$. The only requirement is that each possible key value have a corresponding bin in **B**. The extended Binsort algorithm is shown in Figure 7.16.

This version of Binsort can sort any collection of records whose key values fall in the range from 0 to **MaxKeyValue**-1. The total work required is simply that needed to place each record into the appropriate bin and then take all of the records out of the bins. Thus, we need to process each record twice, for $\Theta(n)$ work.

Unfortunately, there is a crucial oversight in this analysis. Binsort must also look at each of the bins to see if it contains a record. The algorithm must process **MaxKeyValue** bins, regardless of how many actually hold records. If **MaxKeyValue** is small compared to n, then this is not a great expense. Suppose that **MaxKeyValue** $= n^2$. In this case, the total amount of work done will be $\Theta(n + n^2) = \Theta(n^2)$. This results in a poor sorting algorithm, and the algorithm becomes even worse as the disparity between n and **MaxKeyValue** increases. In addition,

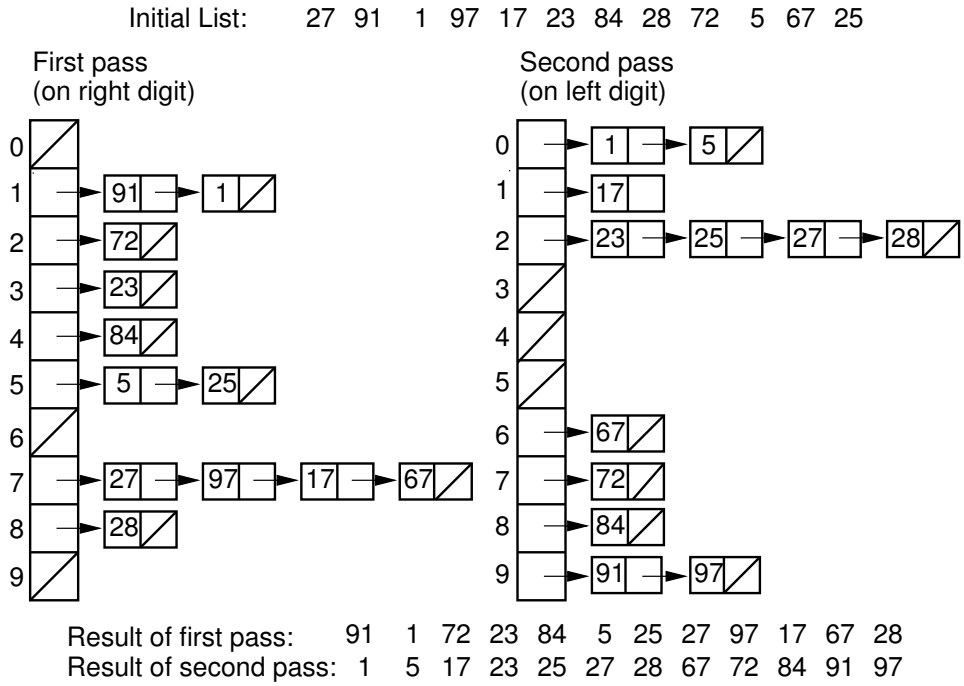

Figure 7.17 An example of Radix Sort for twelve two-digit numbers in base ten. Two passes are required to sort the list.

a large key range requires an unacceptably large array **B**. Thus, even the extended Binsort is useful only for a limited key range.

A further generalization to Binsort yields a **bucket sort**. Each bin is associated with not just one key, but rather a range of key values. A bucket sort assigns records to bins and then relies on some other sorting technique to sort the records within each bin. The hope is that the relatively inexpensive bucketing process will put only a small number of records in each bin, and that a "cleanup sort" within the bins will then be relatively cheap.

There is a way to keep the number of bins and the related processing small while allowing the cleanup sort to be based on Binsort. Consider a sequence of records with keys in the range 0 to 99. If we have ten bins available, we can first assign records to bins by taking their key value modulo 10. Thus, every key will be assigned to the bin matching its rightmost decimal digit. We can then take these records from the bins *in order* and reassign them to the bins on the basis of their leftmost (10's place) digit (define values in the range 0 to 9 to have a leftmost digit of 0). In other words, assign the ith record from array **A** to a bin using the formula **A[i]/10**. If we now gather the values from the bins in order, the result is a sorted list. Figure 7.17 illustrates this process.

```
template <typename E, typename getKey>
void radix(E A[], E B[],
           int n, int k, int r, int cnt[]) {
  // cnt[i] stores number of records in bin[i]
  int j;

  for (int i=0, rtoi=1; i<k; i++, rtoi*=r) { // For k digits
    for (j=0; j<r; j++) cnt[j] = 0;        // Initialize cnt

    // Count the number of records for each bin on this pass
    for (j=0; j<n; j++) cnt[(getKey::key(A[j])/rtoi)%r]++;

    // Index B: cnt[j] will be index for last slot of bin j.
    for (j=1; j<r; j++) cnt[j] = cnt[j-1] + cnt[j];

    // Put records into bins, work from bottom of each bin.
    // Since bins fill from bottom, j counts downwards
    for (j=n-1; j>=0; j--)
      B[--cnt[(getKey::key(A[j])/rtoi)%r]] = A[j];

    for (j=0; j<n; j++) A[j] = B[j];    // Copy B back to A
  }
}
```

Figure 7.18 The Radix Sort algorithm.

In this example, we have $r = 10$ bins and $n = 12$ keys in the range 0 to $r^2 - 1$. The total computation is $\Theta(n)$, because we look at each record and each bin a constant number of times. This is a great improvement over the simple Binsort where the number of bins must be as large as the key range. Note that the example uses $r = 10$ so as to make the bin computations easy to visualize: Records were placed into bins based on the value of first the rightmost and then the leftmost decimal digits. Any number of bins would have worked. This is an example of a **Radix Sort**, so called because the bin computations are based on the **radix** or the **base** of the key values. This sorting algorithm can be extended to any number of keys in any key range. We simply assign records to bins based on the keys' digit values working from the rightmost digit to the leftmost. If there are k digits, then this requires that we assign keys to bins k times.

As with Mergesort, an efficient implementation of Radix Sort is somewhat difficult to achieve. In particular, we would prefer to sort an array of values and avoid processing linked lists. If we know how many values will be in each bin, then an auxiliary array of size r can be used to hold the bins. For example, if during the first pass the 0 bin will receive three records and the 1 bin will receive five records, then we could simply reserve the first three array positions for the 0 bin and the next five array positions for the 1 bin. Exactly this approach is taken by the **C**++ implementation of Figure 7.18. At the end of each pass, the records are copied back to the original array.

The first inner **for** loop initializes array **cnt**. The second loop counts the number of records to be assigned to each bin. The third loop sets the values in **cnt** to their proper indices within array **B**. Note that the index stored in **cnt[j]** is the *last* index for bin **j**; bins are filled from high index to low index. The fourth loop assigns the records to the bins (within array **B**). The final loop simply copies the records back to array **A** to be ready for the next pass. Variable **rtoi** stores r^i for use in bin computation on the i'th iteration. Figure 7.19 shows how this algorithm processes the input shown in Figure 7.17.

This algorithm requires k passes over the list of n numbers in base r, with $\Theta(n + r)$ work done at each pass. Thus the total work is $\Theta(nk + rk)$. What is this in terms of n? Because r is the size of the base, it might be rather small. One could use base 2 or 10. Base 26 would be appropriate for sorting character strings. For now, we will treat r as a constant value and ignore it for the purpose of determining asymptotic complexity. Variable k is related to the key range: It is the maximum number of digits that a key may have in base r. In some applications we can determine k to be of limited size and so might wish to consider it a constant. In this case, Radix Sort is $\Theta(n)$ in the best, average, and worst cases, making it the sort with best asymptotic complexity that we have studied.

Is it a reasonable assumption to treat k as a constant? Or is there some relationship between k and n? If the key range is limited and duplicate key values are common, there might be no relationship between k and n. To make this distinction clear, use N to denote the number of distinct key values used by the n records. Thus, $N \leq n$. Because it takes a minimum of $\log_r N$ base r digits to represent N distinct key values, we know that $k \geq \log_r N$.

Now, consider the situation in which no keys are duplicated. If there are n unique keys ($n = N$), then it requires n distinct code values to represent them. Thus, $k \geq \log_r n$. Because it requires *at least* $\Omega(\log n)$ digits (within a constant factor) to distinguish between the n distinct keys, k is in $\Omega(\log n)$. This yields an asymptotic complexity of $\Omega(n \log n)$ for Radix Sort to process n distinct key values.

It is possible that the key range is much larger; $\log_r n$ bits is merely the best case possible for n distinct values. Thus, the $\log_r n$ estimate for k could be overly optimistic. The moral of this analysis is that, for the general case of n distinct key values, Radix Sort is at best a $\Omega(n \log n)$ sorting algorithm.

Radix Sort can be much improved by making base r be as large as possible. Consider the case of an integer key value. Set $r = 2^i$ for some i. In other words, the value of r is related to the number of bits of the key processed on each pass. Each time the number of bits is doubled, the number of passes is cut in half. When processing an integer key value, setting $r = 256$ allows the key to be processed one byte at a time. Processing a 32-bit key requires only four passes. It is not unreasonable on most computers to use $r = 2^{16} = 64$K, resulting in only two passes for

Figure 7.19 An example showing function **`radix`** applied to the input of Figure 7.17. Row 1 shows the initial values within the input array. Row 2 shows the values for array **`cnt`** after counting the number of records for each bin. Row 3 shows the index values stored in array **`cnt`**. For example, **`cnt[0]`** is 0, indicating no input values are in bin 0. **`Cnt[1]`** is 2, indicating that array **`B`** positions 0 and 1 will hold the values for bin 1. **`Cnt[2]`** is 3, indicating that array **`B`** position 2 will hold the (single) value for bin 2. **`Cnt[7]`** is 11, indicating that array **`B`** positions 7 through 10 will hold the four values for bin 7. Row 4 shows the results of the first pass of the Radix Sort. Rows 5 through 7 show the equivalent steps for the second pass.

a 32-bit key. Of course, this requires a **cnt** array of size 64K. Performance will be good only if the number of records is close to 64K or greater. In other words, the number of records must be large compared to the key size for Radix Sort to be efficient. In many sorting applications, Radix Sort can be tuned in this way to give good performance.

Radix Sort depends on the ability to make a fixed number of multiway choices based on a digit value, as well as random access to the bins. Thus, Radix Sort might be difficult to implement for certain key types. For example, if the keys are real numbers or arbitrary length strings, then some care will be necessary in implementation. In particular, Radix Sort will need to be careful about deciding when the "last digit" has been found to distinguish among real numbers, or the last character in variable length strings. Implementing the concept of Radix Sort with the trie data structure (Section 13.1) is most appropriate for these situations.

At this point, the perceptive reader might begin to question our earlier assumption that key comparison takes constant time. If the keys are "normal integer" values stored in, say, an integer variable, what is the size of this variable compared to n? In fact, it is almost certain that 32 (the number of bits in a standard **int** variable) is greater than $\log n$ for any practical computation. In this sense, comparison of two long integers requires $\Omega(\log n)$ work.

Computers normally do arithmetic in units of a particular size, such as a 32-bit word. Regardless of the size of the variables, comparisons use this native word size and require a constant amount of time since the comparison is implemented in hardware. In practice, comparisons of two 32-bit values take constant time, even though 32 is much greater than $\log n$. To some extent the truth of the proposition that there are constant time operations (such as integer comparison) is in the eye of the beholder. At the gate level of computer architecture, individual bits are compared. However, constant time comparison for integers is true in practice on most computers (they require a fixed number of machine instructions), and we rely on such assumptions as the basis for our analyses. In contrast, Radix Sort must do several arithmetic calculations on key values (each requiring constant time), where the number of such calculations is proportional to the key length. Thus, Radix Sort truly does $\Omega(n \log n)$ work to process n distinct key values.

7.8 An Empirical Comparison of Sorting Algorithms

Which sorting algorithm is fastest? Asymptotic complexity analysis lets us distinguish between $\Theta(n^2)$ and $\Theta(n \log n)$ algorithms, but it does not help distinguish between algorithms with the same asymptotic complexity. Nor does asymptotic analysis say anything about which algorithm is best for sorting small lists. For answers to these questions, we can turn to empirical testing.

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58
Shell	.00034	.008	0.14	1.99	30.2	554	0.44	0.79
Shell/O	.00034	.008	0.12	1.91	29.0	530	0.36	0.64
Merge	.00050	.010	0.12	1.61	19.3	219	0.83	0.79
Merge/O	.00024	.007	0.10	1.31	17.2	197	0.47	0.66
Quick	.00048	.008	0.11	1.37	15.7	162	0.37	0.40
Quick/O	.00031	.006	0.09	1.14	13.6	143	0.32	0.36
Heap	.00050	.011	0.16	2.08	26.7	391	1.57	1.56
Heap/O	.00033	.007	0.11	1.61	20.8	334	1.01	1.04
Radix/4	.00838	.081	0.79	7.99	79.9	808	7.97	7.97
Radix/8	.00799	.044	0.40	3.99	40.0	404	4.00	3.99

Figure 7.20 Empirical comparison of sorting algorithms run on a 3.4-GHz Intel Pentium 4 CPU running Linux. Shellsort, Quicksort, Mergesort, and Heapsort each are shown with regular and optimized versions. Radix Sort is shown for 4- and 8-bit-per-pass versions. All times shown are milliseconds.

Figure 7.20 shows timing results for actual implementations of the sorting algorithms presented in this chapter. The algorithms compared include Insertion Sort, Bubble Sort, Selection Sort, Shellsort, Quicksort, Mergesort, Heapsort and Radix Sort. Shellsort shows both the basic version from Section 7.3 and another with increments based on division by three. Mergesort shows both the basic implementation from Section 7.4 and the optimized version (including calls to Insertion Sort for lists of length below nine). For Quicksort, two versions are compared: the basic implementation from Section 7.5 and an optimized version that does not partition sublists below length nine (with Insertion Sort performed at the end). The first Heapsort version uses the class definitions from Section 5.5. The second version removes all the class definitions and operates directly on the array using inlined code for all access functions.

Except for the rightmost columns, the input to each algorithm is a random array of integers. This affects the timing for some of the sorting algorithms. For example, Selection Sort is not being used to best advantage because the record size is small, so it does not get the best possible showing. The Radix Sort implementation certainly takes advantage of this key range in that it does not look at more digits than necessary. On the other hand, it was not optimized to use bit shifting instead of division, even though the bases used would permit this.

The various sorting algorithms are shown for lists of sizes 10, 100, 1000, 10,000, 100,000, and 1,000,000. The final two columns of each table show the performance for the algorithms on inputs of size 10,000 where the numbers are in ascending (sorted) and descending (reverse sorted) order, respectively. These columns demonstrate best-case performance for some algorithms and worst-case

performance for others. They also show that for some algorithms, the order of input has little effect.

These figures show a number of interesting results. As expected, the $O(n^2)$ sorts are quite poor performers for large arrays. Insertion Sort is by far the best of this group, unless the array is already reverse sorted. Shellsort is clearly superior to any of these $O(n^2)$ sorts for lists of even 100 elements. Optimized Quicksort is clearly the best overall algorithm for all but lists of 10 elements. Even for small arrays, optimized Quicksort performs well because it does one partition step before calling Insertion Sort. Compared to the other $O(n \log n)$ sorts, unoptimized Heapsort is quite slow due to the overhead of the class structure. When all of this is stripped away and the algorithm is implemented to manipulate an array directly, it is still somewhat slower than mergesort. In general, optimizing the various algorithms makes a noticeable improvement for larger array sizes.

Overall, Radix Sort is a surprisingly poor performer. If the code had been tuned to use bit shifting of the key value, it would likely improve substantially; but this would seriously limit the range of element types that the sort could support.

7.9 Lower Bounds for Sorting

This book contains many analyses for algorithms. These analyses generally define the upper and lower bounds for algorithms in their worst and average cases. For many of the algorithms presented so far, analysis has been easy. This section considers a more difficult task — an analysis for the cost of a *problem* as opposed to an *algorithm*. The upper bound for a problem can be defined as the asymptotic cost of the fastest known algorithm. The lower bound defines the best possible efficiency for *any* algorithm that solves the problem, including algorithms not yet invented. Once the upper and lower bounds for the problem meet, we know that no future algorithm can possibly be (asymptotically) more efficient.

A simple estimate for a problem's lower bound can be obtained by measuring the size of the input that must be read and the output that must be written. Certainly no algorithm can be more efficient than the problem's I/O time. From this we see that the sorting problem cannot be solved by *any* algorithm in less than $\Omega(n)$ time because it takes at least n steps to read and write the n values to be sorted. Alternatively, any sorting algorithm must at least look at every input vale to recognize whether the input values are in sort order. So, based on our current knowledge of sorting algorithms and the size of the input, we know that the *problem* of sorting is bounded by $\Omega(n)$ and $O(n \log n)$.

Computer scientists have spent much time devising efficient general-purpose sorting algorithms, but no one has ever found one that is faster than $O(n \log n)$ in the worst or average cases. Should we keep searching for a faster sorting algorithm?

Or can we prove that there is no faster sorting algorithm by finding a tighter lower bound?

This section presents one of the most important and most useful proofs in computer science: No sorting algorithm based on key comparisons can possibly be faster than $\Omega(n \log n)$ in the worst case. This proof is important for three reasons. First, knowing that widely used sorting algorithms are asymptotically optimal is reassuring. In particular, it means that you need not bang your head against the wall searching for an $O(n)$ sorting algorithm (or at least not one in any way based on key comparisons). Second, this proof is one of the few non-trivial lower-bounds proofs that we have for any problem; that is, this proof provides one of the relatively few instances where our lower bound is tighter than simply measuring the size of the input and output. As such, it provides a useful model for proving lower bounds on other problems. Finally, knowing a lower bound for sorting gives us a lower bound in turn for other problems whose solution could be used as the basis for a sorting algorithm. The process of deriving asymptotic bounds for one problem from the asymptotic bounds of another is called a **reduction**, a concept further explored in Chapter 17.

Except for the Radix Sort and Binsort, all of the sorting algorithms presented in this chapter make decisions based on the direct comparison of two key values. For example, Insertion Sort sequentially compares the value to be inserted into the sorted list until a comparison against the next value in the list fails. In contrast, Radix Sort has no direct comparison of key values. All decisions are based on the value of specific digits in the key value, so it is possible to take approaches to sorting that do not involve key comparisons. Of course, Radix Sort in the end does not provide a more efficient sorting algorithm than comparison-based sorting. Thus, empirical evidence suggests that comparison-based sorting is a good approach.[3]

The proof that any comparison sort requires $\Omega(n \log n)$ comparisons in the worst case is structured as follows. First, comparison-based decisions can be modeled as the branches in a tree. This means that any sorting algorithm based on comparisons between records can be viewed as a binary tree whose nodes correspond to the comparisons, and whose branches correspond to the possible outcomes. Next, the minimum number of leaves in the resulting tree is shown to be the factorial of n. Finally, the minimum depth of a tree with $n!$ leaves is shown to be in $\Omega(n \log n)$.

Before presenting the proof of an $\Omega(n \log n)$ lower bound for sorting, we first must define the concept of a **decision tree**. A decision tree is a binary tree that can model the processing for any algorithm that makes binary decisions. Each (binary) decision is represented by a branch in the tree. For the purpose of modeling sorting algorithms, we count all comparisons of key values as decisions. If two keys are

[3]The truth is stronger than this statement implies. In reality, Radix Sort relies on comparisons as well and so can be modeled by the technique used in this section. The result is an $\Omega(n \log n)$ bound in the general case even for algorithms that look like Radix Sort.

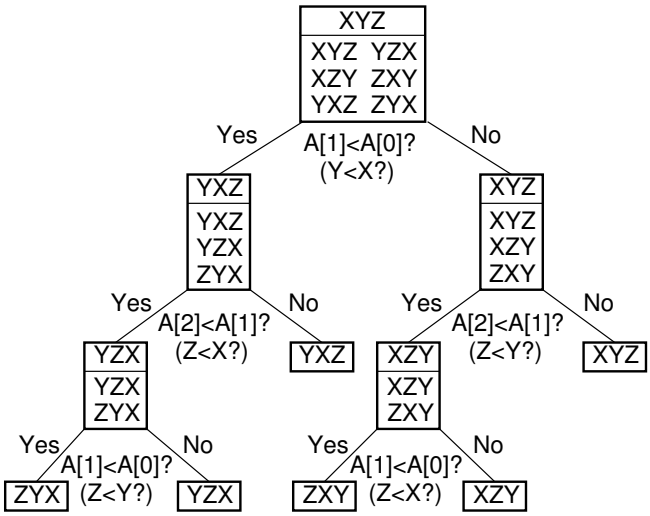

Figure 7.21 Decision tree for Insertion Sort when processing three values labeled X, Y, and Z, initially stored at positions 0, 1, and 2, respectively, in input array A.

compared and the first is less than the second, then this is modeled as a left branch in the decision tree. In the case where the first value is greater than the second, the algorithm takes the right branch.

Figure 7.21 shows the decision tree that models Insertion Sort on three input values. The first input value is labeled X, the second Y, and the third Z. They are initially stored in positions 0, 1, and 2, respectively, of input array **A**. Consider the possible outputs. Initially, we know nothing about the final positions of the three values in the sorted output array. The correct output could be any permutation of the input values. For three values, there are $n! = 6$ permutations. Thus, the root node of the decision tree lists all six permutations that might be the eventual result of the algorithm.

When $n = 3$, the first comparison made by Insertion Sort is between the second item in the input array (Y) and the first item in the array (X). There are two possibilities: Either the value of Y is less than that of X, or the value of Y is *not* less than that of X. This decision is modeled by the first branch in the tree. If Y is less than X, then the left branch should be taken and Y must appear before X in the final output. Only three of the original six permutations have this property, so the left child of the root lists the three permutations where Y appears before X: YXZ, YZX, and ZYX. Likewise, if Y were not less than X, then the right branch would be taken, and only the three permutations in which Y appears after X are possible outcomes: XYZ, XZY, and ZXY. These are listed in the right child of the root.

Let us assume for the moment that Y is less than X and so the left branch is taken. In this case, Insertion Sort swaps the two values. At this point the array

stores YXZ. Thus, in Figure 7.21 the left child of the root shows YXZ above the line. Next, the third value in the array is compared against the second (i.e., Z is compared with X). Again, there are two possibilities. If Z is less than X, then these items should be swapped (the left branch). If Z is not less than X, then Insertion Sort is complete (the right branch).

Note that the right branch reaches a leaf node, and that this leaf node contains only one permutation: YXZ. This means that only permutation YXZ can be the outcome based on the results of the decisions taken to reach this node. In other words, Insertion Sort has "found" the single permutation of the original input that yields a sorted list. Likewise, if the second decision resulted in taking the left branch, a third comparison, regardless of the outcome, yields nodes in the decision tree with only single permutations. Again, Insertion Sort has "found" the correct permutation that yields a sorted list.

Any sorting algorithm based on comparisons can be modeled by a decision tree in this way, regardless of the size of the input. Thus, all sorting algorithms can be viewed as algorithms to "find" the correct permutation of the input that yields a sorted list. Each algorithm based on comparisons can be viewed as proceeding by making branches in the tree based on the results of key comparisons, and each algorithm can terminate once a node with a single permutation has been reached.

How is the worst-case cost of an algorithm expressed by the decision tree? The decision tree shows the decisions made by an algorithm for all possible inputs of a given size. Each path through the tree from the root to a leaf is one possible series of decisions taken by the algorithm. The depth of the deepest node represents the longest series of decisions required by the algorithm to reach an answer.

There are many comparison-based sorting algorithms, and each will be modeled by a different decision tree. Some decision trees might be well-balanced, others might be unbalanced. Some trees will have more nodes than others (those with more nodes might be making "unnecessary" comparisons). In fact, a poor sorting algorithm might have an arbitrarily large number of nodes in its decision tree, with leaves of arbitrary depth. There is no limit to how slow the "worst" possible sorting algorithm could be. However, we are interested here in knowing what the *best* sorting algorithm could have as its minimum cost in the worst case. In other words, we would like to know what is the *smallest* depth possible for the *deepest* node in the tree for any sorting algorithm.

The smallest depth of the deepest node will depend on the number of nodes in the tree. Clearly we would like to "push up" the nodes in the tree, but there is limited room at the top. A tree of height 1 can only store one node (the root); the tree of height 2 can store three nodes; the tree of height 3 can store seven nodes, and so on.

Here are some important facts worth remembering:

- A binary tree of height n can store at most $2^n - 1$ nodes.

- Equivalently, a tree with n nodes requires at least $\lceil \log(n+1) \rceil$ levels.

What is the minimum number of nodes that must be in the decision tree for any comparison-based sorting algorithm for n values? Because sorting algorithms are in the business of determining which unique permutation of the input corresponds to the sorted list, the decision tree for any sorting algorithm must contain at least one leaf node for each possible permutation. There are $n!$ permutations for a set of n numbers (see Section 2.2).

Because there are at least $n!$ nodes in the tree, we know that the tree must have $\Omega(\log n!)$ levels. From Stirling's approximation (Section 2.2), we know $\log n!$ is in $\Omega(n \log n)$. The decision tree for any comparison-based sorting algorithm must have nodes $\Omega(n \log n)$ levels deep. Thus, in the worst case, any such sorting algorithm must require $\Omega(n \log n)$ comparisons.

Any sorting algorithm requiring $\Omega(n \log n)$ comparisons in the worst case requires $\Omega(n \log n)$ running time in the worst case. Because any sorting algorithm requires $\Omega(n \log n)$ running time, the problem of sorting also requires $\Omega(n \log n)$ time. We already know of sorting algorithms with $\mathrm{O}(n \log n)$ running time, so we can conclude that the problem of sorting requires $\Theta(n \log n)$ time. As a corollary, we know that no comparison-based sorting algorithm can improve on existing $\Theta(n \log n)$ time sorting algorithms by more than a constant factor.

7.10 Further Reading

The definitive reference on sorting is Donald E. Knuth's *Sorting and Searching* [Knu98]. A wealth of details is covered there, including optimal sorts for small size n and special purpose sorting networks. It is a thorough (although somewhat dated) treatment on sorting. For an analysis of Quicksort and a thorough survey on its optimizations, see Robert Sedgewick's *Quicksort* [Sed80]. Sedgewick's *Algorithms* [Sed11] discusses most of the sorting algorithms described here and pays special attention to efficient implementation. The optimized Mergesort version of Section 7.4 comes from Sedgewick.

While $\Omega(n \log n)$ is the theoretical lower bound in the worst case for sorting, many times the input is sufficiently well ordered that certain algorithms can take advantage of this fact to speed the sorting process. A simple example is Insertion Sort's best-case running time. Sorting algorithms whose running time is based on the amount of disorder in the input are called **adaptive**. For more information on adaptive sorting algorithms, see "A Survey of Adaptive Sorting Algorithms" by Estivill-Castro and Wood [ECW92].