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 requiring that their type extend the **`Comparable`** class. This will ensure that the class implements the **`compareTo`** method, which returns a value less than zero, equal to zero, or greater than zero depending on its relationship to the record being compared to. The **`compareTo`** method is defined to extract the appropriate key field from the record. We also assume that for every record type there is a **`swap`** function that can interchange the contents of two records in the array.

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

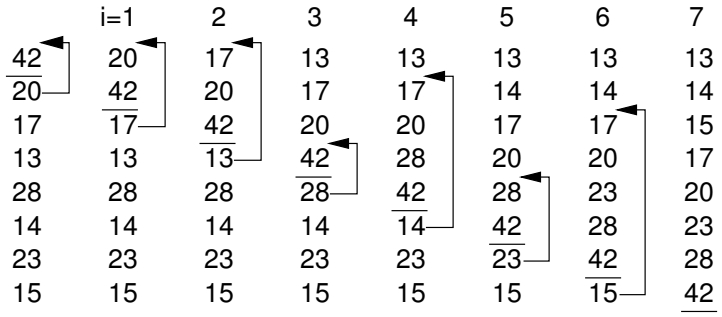

Figure 7.1 An illustration of Insertion Sort. Each column shows the array after the iteration with the indicated value of **i** in the outer **for** loop. Values above the line in each column have been sorted. Arrows indicate the upward motions of records through the array.

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

```
static <E extends Comparable<? super E>>
void inssort(E[] A) {
  for (int i=1; i<A.length; i++) // Insert i'th record
    for (int j=i; (j>0) && (A[j].compareTo(A[j-1])<0); j--)
      DSutil.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

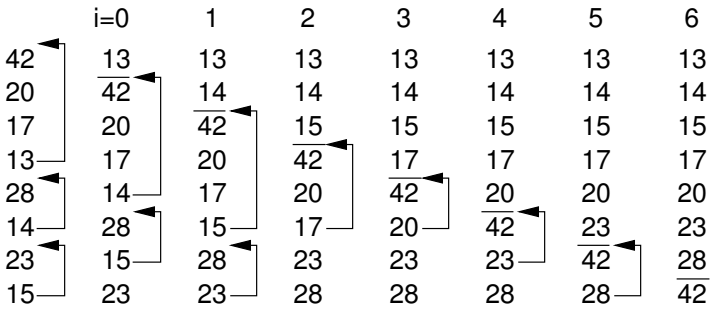

Figure 7.2 An illustration of Bubble Sort. Each column shows the array after the iteration with the indicated value of **i** in the outer **for** loop. Values above the line in each column have been sorted. Arrows indicate the swaps that take place during a given iteration.

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 Java implementation is as follows:

```
static <E extends Comparable<? super E>>
void bubblesort(E[] A) {
  for (int i=0; i<A.length-1; i++) // Bubble up i'th record
    for (int j=A.length-1; j>i; j--)
      if ((A[j].compareTo(A[j-1]) < 0))
        DSutil.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.

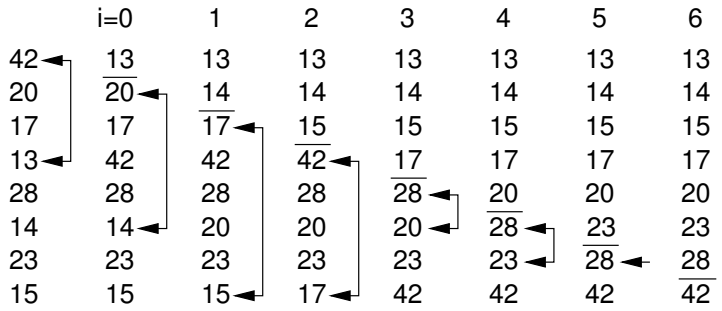

Figure 7.3 An example of Selection Sort. Each column shows the array after the iteration with the indicated value of `i` in the outer `for` loop. Numbers above the line in each column have been sorted and are in their final positions.

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 Java implementation.

```
static <E extends Comparable<? super E>>
void selectsort(E[] A) {
  for (int i=0; i<A.length-1; i++) { // Select i'th record
    int lowindex = i;                // Remember its index
    for (int j=A.length-1; j>i; j--) // Find the least value
      if (A[j].compareTo(A[lowindex]) < 0)
        lowindex = j;                // Put it in place
    DSutil.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

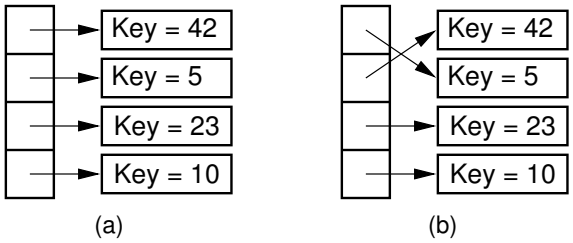

Figure 7.4 An example of swapping pointers to records. (a) A series of four records. The record with key value 42 comes before the record with key value 5. (b) The four records after the top two pointers have been swapped. Now the record with key value 5 comes before the record with key value 42.

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 $n/2$

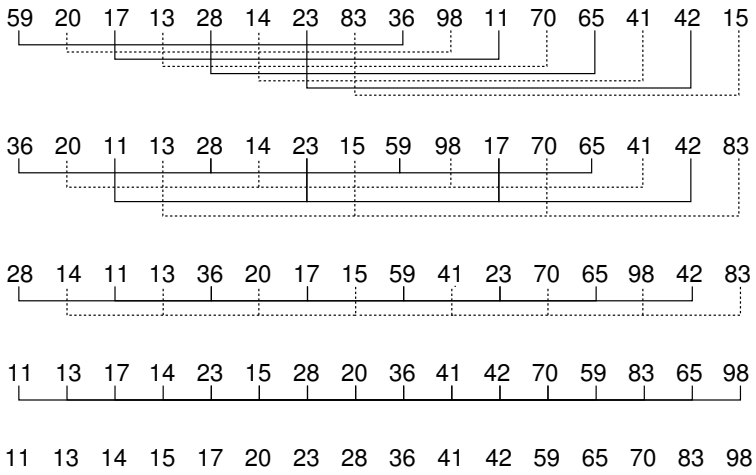

Figure 7.6 An example of Shellsort. Sixteen items are sorted in four passes. The first pass sorts 8 sublists of size 2 and increment 8. The second pass sorts 4 sublists of size 4 and increment 4. The third pass sorts 2 sublists of size 8 and increment 2. The fourth pass sorts 1 list of size 16 and increment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

```
static <E extends Comparable<? super E>>
void shellsort(E[] A) {
  for (int i=A.length/2; i>2; i/=2) // For each increment
    for (int j=0; j<i; j++)         // Sort each sublist
      inssort2(A, j, i);
  inssort2(A, 0, 1);     // Could call regular inssort here
}

/** Modified Insertion Sort for varying increments */
static <E extends Comparable<? super E>>
void inssort2(E[] A, int start, int incr) {
  for (int i=start+incr; i<A.length; i+=incr)
    for (int j=i; (j>=incr)&&
                  (A[j].compareTo(A[j-incr])<0); j-=incr)
      DSutil.swap(A, j, j-incr);
}
```

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**.

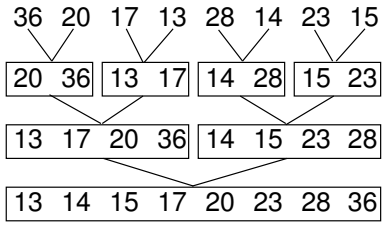

Figure 7.8 An illustration of Mergesort. The first row shows eight numbers that are to be sorted. Mergesort will recursively subdivide the list into sublists of one element each, then recombine the sublists. The second row shows the four sublists of size 2 created by the first merging pass. The third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

```
static <E extends Comparable<? super E>>
void mergesort(E[] A, E[] temp, int l, int r) {
  int mid = (l+r)/2;                // Select midpoint
  if (l == r) return;               // List has one element
  mergesort(A, temp, l, mid);   // Mergesort first half
  mergesort(A, temp, mid+1, r); // Mergesort second half
  for (int i=l; i<=r; i++)          // Copy subarray to temp
    temp[i] = A[i];
  // Do the merge operation back to A
  int i1 = l; int i2 = mid + 1;
  for (int curr=l; curr<=r; curr++) {
    if (i1 == mid+1)              // Left sublist exhausted
      A[curr] = temp[i2++];
    else if (i2 > r)              // Right sublist exhausted
      A[curr] = temp[i1++];
    else if (temp[i1].compareTo(temp[i2])<0) // Get smaller
      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

```
static <E extends Comparable<? super E>>
void mergesort(E[] A, E[] temp, int l, int r) {
  int i, j, k, mid = (l+r)/2;  // Select the midpoint
  if (l == r) return;          // List has one element
  if ((mid-l) >= THRESHOLD) mergesort(A, temp, l, mid);
  else inssort(A, l, mid-l+1);
  if ((r-mid) > THRESHOLD) mergesort(A, temp, mid+1, r);
  else inssort(A, mid+1, r-mid);
  // Do the merge operation.  First, copy 2 halves to temp.
  for (i=l; i<=mid; i++) temp[i] = A[i];
  for (j=1; j<=r-mid; j++) temp[r-j+1] = A[j+mid];
  // Merge sublists back to array
  for (i=l,j=r,k=l; k<=r; k++)
    if (temp[i].compareTo(temp[j])<0) 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 Java 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

```
static <E extends Comparable<? super E>>
void qsort(E[] A, int i, int j) {      // Quicksort
  int pivotindex = findpivot(A, i, j); // Pick a pivot
  DSutil.swap(A, pivotindex, j);       // Stick pivot at end
  // k will be the first position in the right subarray
  int k = partition(A, i-1, j, A[j]);
  DSutil.swap(A, k, j);                // Put pivot in place
  if ((k-i) > 1) qsort(A, i, k-1);     // Sort left partition
  if ((j-k) > 1) qsort(A, k+1, j);     // Sort right partition
}
```

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:

```
static <E extends Comparable<? super E>>
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 Java 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** loop. This ensures that **r** does not run off the low end of the partition in the case

```
static <E extends Comparable<? super E>>
int partition(E[] A, int l, int r, E pivot) {
  do {                  // Move bounds inward until they meet
    while (A[++l].compareTo(pivot)<0);
    while ((r!=0) && (A[--r].compareTo(pivot)>0));
    DSutil.swap(A, l, r);       // Swap out-of-place values
  } while (l < r);              // Stop when they cross
  DSutil.swap(A, l, r);         // Reverse last, wasted swap
  return l;      // Return first position in right partition
}
```

Figure 7.12 The Quicksort partition implementation.

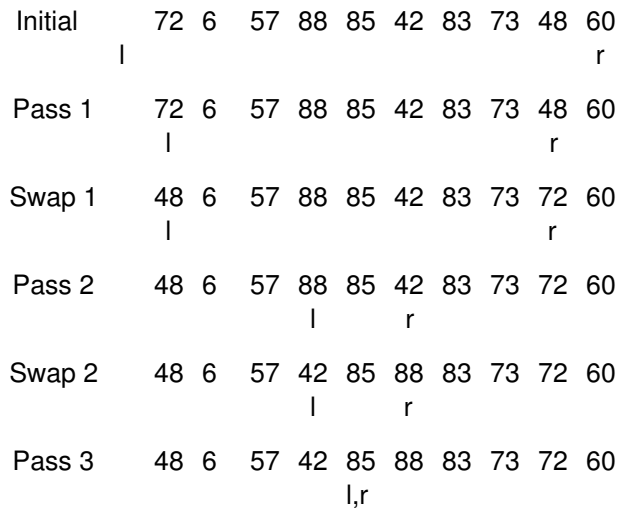

Figure 7.13 The Quicksort partition step. The first row shows the initial positions for a collection of ten key values. The pivot value is 60, which has been swapped to the end of the array. The **do** loop makes three iterations, each time moving counters **l** and **r** inwards until they meet in the 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.

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 total of s steps for a subarray of length s. However, this does not directly tell us

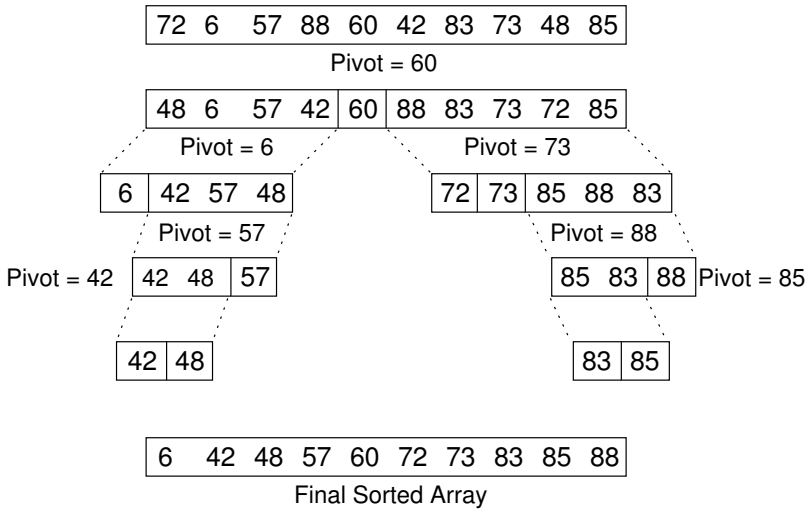

Figure 7.14 An illustration of Quicksort.

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 Java implementation is as follows:

```
static <E extends Comparable<? super E>>
void heapsort(E[] A) {
  // The heap constructor invokes the buildheap method
  MaxHeap<E> H = new MaxHeap<E>(A, A.length, A.length);
  for (int i=0; i<A.length; i++)  // Now sort
    H.removemax(); // Removemax places max at end of heap
}
```

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];
```

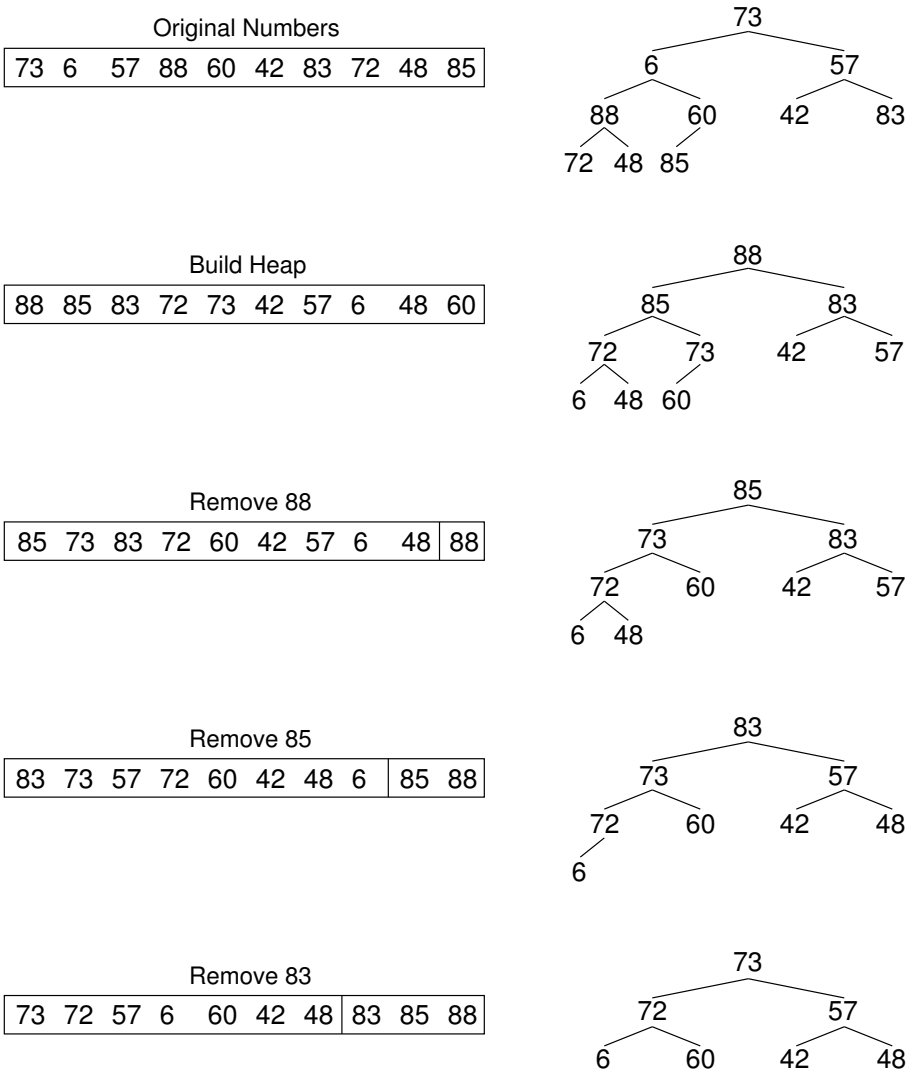

Figure 7.15 An illustration of Heapsort. The top row shows the values in their original order. The second row shows the values after building the heap. The third row shows the result of the first **`removefirst`** operation on key value 88. Note that 88 is now at the end of the array. The fourth row 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.

```
static void binsort(Integer A[]) {
  List<Integer>[] B = (LList<Integer>[])new LList[MaxKey];
  Integer item;
  for (int i=0; i<MaxKey; i++)
    B[i] = new LList<Integer>();
  for (int i=0; i<A.length; i++) B[A[i]].append(A[i]);
  for (int i=0; i<MaxKey; i++)
    for (B[i].moveToStart();
         (item = B[i].getValue()) != null; 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.

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,

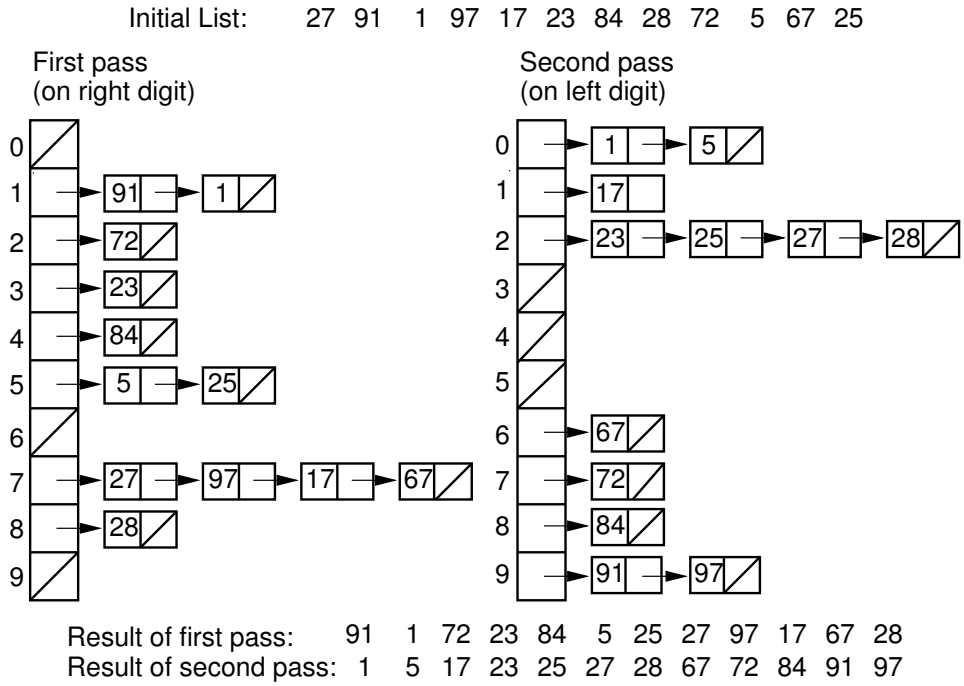

Figure 7.17 An example of Radix Sort for twelve two-digit numbers in base ten. Two passes are required to sort the list.

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.

```
static void radix(Integer[] A, Integer[] B,
                  int k, int r, int[] count) {
  // Count[i] stores number of records in bin[i]
  int i, j, rtok;

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

    // Count the number of records for each bin on this pass
    for (j=0; j<A.length; j++) count[(A[j]/rtok)%r]++;

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

    // Put records into bins, working from bottom of bin
    // Since bins fill from bottom, j counts downwards
    for (j=A.length-1; j>=0; j--)
      B[--count[(A[j]/rtok)%r]] = A[j];

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

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 Java 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.

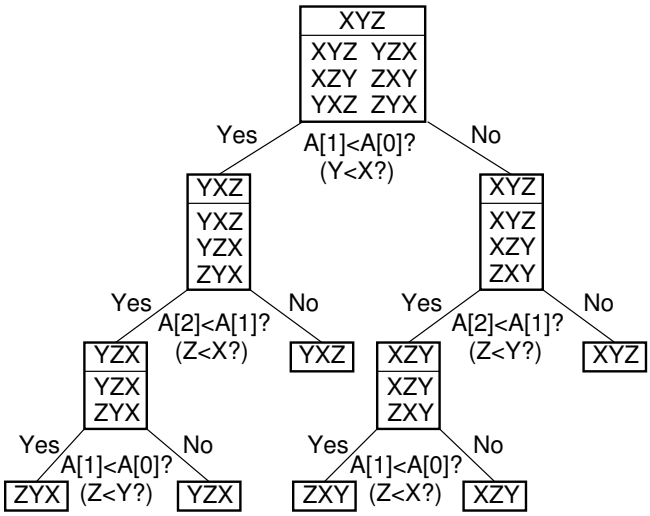

Figure 7.21 Decision tree for Insertion Sort when processing three values labeled X, Y, and Z, initially stored at positions 0, 1, and 2, respectively, in input array A.

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].

7.11 Exercises

7.1 Using induction, prove that Insertion Sort will always produce a sorted array.

7.2 Write an Insertion Sort algorithm for integer key values. However, here's the catch: The input is a stack (*not* an array), and the only variables that your algorithm may use are a fixed number of integers and a fixed number of stacks. The algorithm should return a stack containing the records in sorted order (with the least value being at the top of the stack). Your algorithm should be $\Theta(n^2)$ in the worst case.

7.3 The Bubble Sort implementation has the following inner **`for`** loop:

```
for (int j=n-1; j>i; j--)
```

Consider the effect of replacing this with the following statement:

```
for (int j=n-1; j>0; j--)
```

Would the new implementation work correctly? Would the change affect the asymptotic complexity of the algorithm? How would the change affect the running time of the algorithm?

7.4 When implementing Insertion Sort, a binary search could be used to locate the position within the first $i-1$ elements of the array into which element i should be inserted. How would this affect the number of comparisons required? How would using such a binary search affect the asymptotic running time for Insertion Sort?

7.5 Figure 7.5 shows the best-case number of swaps for Selection Sort as $\Theta(n)$. This is because the algorithm does not check to see if the ith record is already in the ith position; that is, it might perform unnecessary swaps.

- **(a)** Modify the algorithm so that it does not make unnecessary swaps.
- **(b)** What is your prediction regarding whether this modification actually improves the running time?
- **(c)** Write two programs to compare the actual running times of the original Selection Sort and the modified algorithm. Which one is actually faster?

7.6 Recall that a sorting algorithm is said to be stable if the original ordering for duplicate keys is preserved. Of the sorting algorithms Insertion Sort, Bubble Sort, Selection Sort, Shellsort, Mergesort, Quicksort, Heapsort, Binsort, and Radix Sort, which of these are stable, and which are not? For each one, describe either why it is or is not stable. If a minor change to the implementation would make it stable, describe the change.

7.7 Recall that a sorting algorithm is said to be stable if the original ordering for duplicate keys is preserved. We can make any algorithm stable if we alter the input keys so that (potentially) duplicate key values are made unique in a way that the first occurrence of the original duplicate value is less than the second occurrence, which in turn is less than the third, and so on. In the worst case, it is possible that all n input records have the same key value. Give an

algorithm to modify the key values such that every modified key value is unique, the resulting key values give the same sort order as the original keys, the result is stable (in that the duplicate original key values remain in their original order), and the process of altering the keys is done in linear time using only a constant amount of additional space.

7.8 The discussion of Quicksort in Section 7.5 described using a stack instead of recursion to reduce the number of function calls made.

(a) How deep can the stack get in the worst case?

(b) Quicksort makes two recursive calls. The algorithm could be changed to make these two calls in a specific order. In what order should the two calls be made, and how does this affect how deep the stack can become?

7.9 Give a permutation for the values 0 through 7 that will cause Quicksort (as implemented in Section 7.5) to have its worst case behavior.

7.10 Assume `L` is an array, `L.length()` returns the number of records in the array, and `qsort(L, i, j)` sorts the records of `L` from `i` to `j` (leaving the records sorted in `L`) using the Quicksort algorithm. What is the average-case time complexity for each of the following code fragments?

(a)
```
for (i=0; i<L.length; i++)
   qsort(L, 0, i);
```

(b)
```
for (i=0; i<L.length; i++)
   qsort(L, 0, L.length-1);
```

7.11 Modify Quicksort to find the smallest k values in an array of records. Your output should be the array modified so that the k smallest values are sorted in the first k positions of the array. Your algorithm should do the minimum amount of work necessary, that is, no more of the array than necessary should be sorted.

7.12 Modify Quicksort to sort a sequence of variable-length strings stored one after the other in a character array, with a second array (storing pointers to strings) used to index the strings. Your function should modify the index array so that the first pointer points to the beginning of the lowest valued string, and so on.

7.13 Graph $f_1(n) = n \log n$, $f_2(n) = n^{1.5}$, and $f_3(n) = n^2$ in the range $1 \leq n \leq 1000$ to visually compare their growth rates. Typically, the constant factor in the running-time expression for an implementation of Insertion Sort will be less than the constant factors for Shellsort or Quicksort. How many times greater can the constant factor be for Shellsort to be faster than Insertion Sort when $n = 1000$? How many times greater can the constant factor be for Quicksort to be faster than Insertion Sort when $n = 1000$?

7.14 Imagine that there exists an algorithm **SPLITk** that can split a list **L** of n elements into k sublists, each containing one or more elements, such that sublist i contains only elements whose values are less than all elements in sublist j for $i < j <= k$. If $n < k$, then $k-n$ sublists are empty, and the rest are of length 1. Assume that SPLITk has time complexity O(length of **L**). Furthermore, assume that the k lists can be concatenated again in constant time. Consider the following algorithm:

```
List SORTk(List L) {
  List sub[k]; // To hold the sublists
  if (L.length() > 1) {
    SPLITk(L, sub); // SPLITk places sublists into sub
    for (i=0; i<k; i++)
      sub[i] = SORTk(sub[i]); // Sort each sublist
    L = concatenation of k sublists in sub;
    return L;
  }
}
```

(a) What is the worst-case asymptotic running time for SORTk? Why?
(b) What is the average-case asymptotic running time of SORTk? Why?

7.15 Here is a variation on sorting. The problem is to sort a collection of n nuts and n bolts by size. It is assumed that for each bolt in the collection, there is a corresponding nut of the same size, but initially we do not know which nut goes with which bolt. The differences in size between two nuts or two bolts can be too small to see by eye, so you cannot rely on comparing the sizes of two nuts or two bolts directly. Instead, you can only compare the sizes of a nut and a bolt by attempting to screw one into the other (assume this comparison to be a constant time operation). This operation tells you that either the nut is bigger than the bolt, the bolt is bigger than the nut, or they are the same size. What is the minimum number of comparisons needed to sort the nuts and bolts in the worst case?

7.16 **(a)** Devise an algorithm to sort three numbers. It should make as few comparisons as possible. How many comparisons and swaps are required in the best, worst, and average cases?
(b) Devise an algorithm to sort five numbers. It should make as few comparisons as possible. How many comparisons and swaps are required in the best, worst, and average cases?
(c) Devise an algorithm to sort eight numbers. It should make as few comparisons as possible. How many comparisons and swaps are required in the best, worst, and average cases?

7.17 Devise an efficient algorithm to sort a set of numbers with values in the range 0 to 30,000. There are no duplicates. Keep memory requirements to a minimum.

7.18 Which of the following operations are best implemented by first sorting the list of numbers? For each operation, briefly describe an algorithm to implement it, and state the algorithm's asymptotic complexity.

(a) Find the minimum value.
(b) Find the maximum value.
(c) Compute the arithmetic mean.
(d) Find the median (i.e., the middle value).
(e) Find the mode (i.e., the value that appears the most times).

7.19 Consider a recursive Mergesort implementation that calls Insertion Sort on sublists smaller than some threshold. If there are n calls to Mergesort, how many calls will there be to Insertion Sort? Why?

7.20 Implement Mergesort for the case where the input is a linked list.

7.21 Counting sort (assuming the input key values are integers in the range 0 to $m-1$) works by counting the number of records with each key value in the first pass, and then uses this information to place the records in order in a second pass. Write an implementation of counting sort (see the implementation of radix sort for some ideas). What can we say about the relative values of m and n for this to be effective? If $m < n$, what is the running time of this algorithm?

7.22 Use an argument similar to that given in Section 7.9 to prove that $\log n$ is a worst-case lower bound for the problem of searching for a given value in a sorted array containing n elements.

7.23 A simpler way to do the Quicksort partition step is to set index **`split`** to the position of the first value greater than the pivot. Then from position **`split+1`** have another index **`curr`** move to the right until it finds a value less than a pivot. Swap the values at **`split`** and **`next`**, and increment **`split`**. Continue in this way, swapping the smaller values to the left side. When **`curr`** reaches the end of the subarray, **`split`** will be at the split point between the two partitions. Unfortunately, this approach requires more swaps than does the version presented in Section 7.5, resulting in a slower implementation. Give an example and explain why.

7.12 Projects

7.1 One possible improvement for Bubble Sort would be to add a flag variable and a test that determines if an exchange was made during the current iteration. If no exchange was made, then the list is sorted and so the algorithm can stop early. This makes the best case performance become $\mathrm{O}(n)$ (because if the list is already sorted, then no iterations will take place on the first pass, and the sort will stop right there).

Modify the Bubble Sort implementation to add this flag and test. Compare the modified implementation on a range of inputs to determine if it does or does not improve performance in practice.

7.2 Double Insertion Sort is a variation on Insertion Sort that works from the middle of the array out. At each iteration, some middle portion of the array is sorted. On the next iteration, take the two adjacent elements to the sorted portion of the array. If they are out of order with respect to each other, than swap them. Now, push the left element toward the right in the array so long as it is greater than the element to its right. And push the right element toward the left in the array so long as it is less than the element to its left. The algorithm begins by processing the middle two elements of the array if the array is even. If the array is odd, then skip processing the middle item and begin with processing the elements to its immediate left and right.

First, explain what the cost of Double Insertion Sort will be in comparison to standard Insertion sort, and why. (Note that the two elements being processed in the current iteration, once initially swapped to be sorted with with respect to each other, cannot cross as they are pushed into sorted position.) Then, implement Double Insertion Sort, being careful to properly handle both when the array is odd and when it is even. Compare its running time in practice against standard Insertion Sort. Finally, explain how this speedup might affect the threshold level and running time for a Quicksort implementation.

7.3 Perform a study of Shellsort, using different increments. Compare the version shown in Section 7.3, where each increment is half the previous one, with others. In particular, try implementing "division by 3" where the increments on a list of length n will be $n/3$, $n/9$, etc. Do other increment schemes work as well?

7.4 The implementation for Mergesort given in Section 7.4 takes an array as input and sorts that array. At the beginning of Section 7.4 there is a simple pseudocode implementation for sorting a linked list using Mergesort. Implement both a linked list-based version of Mergesort and the array-based version of Mergesort, and compare their running times.

7.5 Starting with the Java code for Quicksort given in this chapter, write a series of Quicksort implementations to test the following optimizations on a wide range of input data sizes. Try these optimizations in various combinations to try and develop the fastest possible Quicksort implementation that you can.

- **(a)** Look at more values when selecting a pivot.
- **(b)** Do not make a recursive call to `qsort` when the list size falls below a given threshold, and use Insertion Sort to complete the sorting process. Test various values for the threshold size.
- **(c)** Eliminate recursion by using a stack and inline functions.

7.6 It has been proposed that Heapsort can be optimized by altering the heap's siftdown function. Call the value being sifted down X. Siftdown does two comparisons per level: First the children of X are compared, then the winner is compared to X. If X is too small, it is swapped with its larger child and the process repeated. The proposed optimization dispenses with the test against X. Instead, the larger child automatically replaces X, until X reaches the bottom level of the heap. At this point, X might be too large to remain in that position. This is corrected by repeatedly comparing X with its parent and swapping as necessary to "bubble" it up to its proper level. The claim is that this process will save a number of comparisons because most nodes when sifted down end up near the bottom of the tree anyway. Implement both versions of siftdown, and do an empirical study to compare their running times.

7.7 Radix Sort is typically implemented to support only a radix that is a power of two. This allows for a direct conversion from the radix to some number of bits in an integer key value. For example, if the radix is 16, then a 32-bit key will be processed in 8 steps of 4 bits each. This can lead to a more efficient implementation because bit shifting can replace the division operations shown in the implementation of Section 7.7. Re-implement the Radix Sort code given in Section 7.7 to use bit shifting in place of division. Compare the running time of the old and new Radix Sort implementations.

7.8 Write your own collection of sorting programs to implement the algorithms described in this chapter, and compare their running times. Be sure to implement optimized versions, trying to make each program as fast as possible. Do you get the same relative timings as shown in Figure 7.20? If not, why do you think this happened? How do your results compare with those of your classmates? What does this say about the difficulty of doing empirical timing studies?

8

File Processing and External Sorting

Earlier chapters presented basic data structures and algorithms that operate on data stored in main memory. Some applications require that large amounts of information be stored and processed — so much information that it cannot all fit into main memory. In that case, the information must reside on disk and be brought into main memory selectively for processing.

You probably already realize that main memory access is much faster than access to data stored on disk or other storage devices. The relative difference in access times is so great that efficient disk-based programs require a different approach to algorithm design than most programmers are used to. As a result, many programmers do a poor job when it comes to file processing applications.

This chapter presents the fundamental issues relating to the design of algorithms and data structures for disk-based applications.[1] We begin with a description of the significant differences between primary memory and secondary storage. Section 8.2 discusses the physical aspects of disk drives. Section 8.3 presents basic methods for managing buffer pools. Section 8.4 discusses the Java model for random access to data stored on disk. Section 8.5 discusses the basic principles for sorting collections of records too large to fit in main memory.

8.1 Primary versus Secondary Storage

Computer storage devices are typically classified into **primary** or **main** memory and **secondary** or **peripheral** storage. Primary memory usually refers to **Random**

[1] Computer technology changes rapidly. I provide examples of disk drive specifications and other hardware performance numbers that are reasonably up to date as of the time when the book was written. When you read it, the numbers might seem out of date. However, the basic principles do not change. The approximate ratios for time, space, and cost between memory and disk have remained surprisingly steady for over 20 years.

Medium	1996	1997	2000	2004	2006	2008	2011
RAM	$45.00	7.00	1.500	0.3500	0.1500	0.0339	0.0138
Disk	0.25	0.10	0.010	0.0010	0.0005	0.0001	0.0001
USB drive	–	–	–	0.1000	0.0900	0.0029	0.0018
Floppy	0.50	0.36	0.250	0.2500	–	–	–
Tape	0.03	0.01	0.001	0.0003	–	–	–
Solid State	–	–	–	–	–	–	0.0021

Figure 8.1 Price comparison table for some writable electronic data storage media in common use. Prices are in US Dollars/MB.

Access Memory (RAM), while secondary storage refers to devices such as hard disk drives, solid state drives, removable "USB" drives, CDs, and DVDs. Primary memory also includes registers, cache, and video memories, but we will ignore them for this discussion because their existence does not affect the principal differences between primary and secondary memory.

Along with a faster CPU, every new model of computer seems to come with more main memory. As memory size continues to increase, is it possible that relatively slow disk storage will be unnecessary? Probably not, because the desire to store and process larger files grows at least as fast as main memory size. Prices for both main memory and peripheral storage devices have dropped dramatically in recent years, as demonstrated by Figure 8.1. However, the cost per unit of disk drive storage is about two orders of magnitude less than RAM and has been for many years.

There is now a wide range of removable media available for transferring data or storing data offline in relative safety. These include floppy disks (now largely obsolete), writable CDs and DVDs, "flash" drives, and magnetic tape. Optical storage such as CDs and DVDs costs roughly half the price of hard disk drive space per megabyte, and have become practical for use as backup storage within the past few years. Tape used to be much cheaper than other media, and was the preferred means of backup, but are not so popular now as other media have decreased in price. "Flash" drives cost the most per megabyte, but due to their storage capacity and flexibility, quickly replaced floppy disks as the primary storage device for transferring data between computer when direct network transfer is not available.

Secondary storage devices have at least two other advantages over RAM memory. Perhaps most importantly, disk, "flash," and optical media are **persistent**, meaning that they are not erased from the media when the power is turned off. In contrast, RAM used for main memory is usually **volatile** — all information is lost with the power. A second advantage is that CDs and "USB" drives can easily be transferred between computers. This provides a convenient way to take information from one computer to another.

In exchange for reduced storage costs, persistence, and portability, secondary storage devices pay a penalty in terms of increased access time. While not all accesses to disk take the same amount of time (more on this later), the typical time required to access a byte of storage from a disk drive in 2011 is around 9 ms (i.e., 9 *thousandths* of a second). This might not seem slow, but compared to the time required to access a byte from main memory, this is fantastically slow. Typical access time from standard personal computer RAM in 2011 is about 5-10 nanoseconds (i.e., 5-10 *billionths* of a second). Thus, the time to access a byte of data from a disk drive is about six orders of magnitude greater than that required to access a byte from main memory. While disk drive and RAM access times are both decreasing, they have done so at roughly the same rate. The relative speeds have remained the same for over several decades, in that the difference in access time between RAM and a disk drive has remained in the range between a factor of 100,000 and 1,000,000.

To gain some intuition for the significance of this speed difference, consider the time that it might take for you to look up the entry for disk drives in the index of this book, and then turn to the appropriate page. Call this your "primary memory" access time. If it takes you about 20 seconds to perform this access, then an access taking 500,000 times longer would require months.

It is interesting to note that while processing speeds have increased dramatically, and hardware prices have dropped dramatically, disk and memory access times have improved by less than an order of magnitude over the past 15 years. However, the situation is really much better than that modest speedup would suggest. During the same time period, the size of both disk and main memory has increased by over three orders of magnitude. Thus, the access times have actually decreased in the face of a massive increase in the density of these storage devices.

Due to the relatively slow access time for data on disk as compared to main memory, great care is required to create efficient applications that process disk-based information. The million-to-one ratio of disk access time versus main memory access time makes the following rule of paramount importance when designing disk-based applications:

Minimize the number of disk accesses!

There are generally two approaches to minimizing disk accesses. The first is to arrange information so that if you do access data from secondary memory, you will get what you need in as few accesses as possible, and preferably on the first access. **File structure** is the term used for a data structure that organizes data stored in secondary memory. File structures should be organized so as to minimize the required number of disk accesses. The other way to minimize disk accesses is to save information previously retrieved (or retrieve additional data with each access at little additional cost) that can be used to minimize the need for future accesses.

This requires the ability to guess accurately what information will be needed later and store it in primary memory now. This is referred to as **caching**.

8.2 Disk Drives

A Java programmer views a random access file stored on disk as a contiguous series of bytes, with those bytes possibly combining to form data records. This is called the **logical** file. The **physical** file actually stored on disk is usually not a contiguous series of bytes. It could well be in pieces spread all over the disk. The **file manager**, a part of the operating system, is responsible for taking requests for data from a logical file and mapping those requests to the physical location of the data on disk. Likewise, when writing to a particular logical byte position with respect to the beginning of the file, this position must be converted by the file manager into the corresponding physical location on the disk. To gain some appreciation for the the approximate time costs for these operations, you need to understand the physical structure and basic workings of a disk drive.

Disk drives are often referred to as **direct access** storage devices. This means that it takes roughly equal time to access any record in the file. This is in contrast to **sequential access** storage devices such as tape drives, which require the tape reader to process data from the beginning of the tape until the desired position has been reached. As you will see, the disk drive is only approximately direct access: At any given time, some records are more quickly accessible than others.

8.2.1 Disk Drive Architecture

A hard disk drive is composed of one or more round **platters**, stacked one on top of another and attached to a central **spindle**. Platters spin continuously at a constant rate. Each usable surface of each platter is assigned a **read/write head** or **I/O head** through which data are read or written, somewhat like the arrangement of a phonograph player's arm "reading" sound from a phonograph record. Unlike a phonograph needle, the disk read/write head does not actually touch the surface of a hard disk. Instead, it remains slightly above the surface, and any contact during normal operation would damage the disk. This distance is very small, much smaller than the height of a dust particle. It can be likened to a 5000-kilometer airplane trip across the United States, with the plane flying at a height of one meter!

A hard disk drive typically has several platters and several read/write heads, as shown in Figure 8.2(a). Each head is attached to an **arm**, which connects to the **boom**.[2] The boom moves all of the heads in or out together. When the heads are in some position over the platters, there are data on each platter directly accessible

[2] This arrangement, while typical, is not necessarily true for all disk drives. Nearly everything said here about the physical arrangement of disk drives represents a typical engineering compromise, not a fundamental design principle. There are many ways to design disk drives, and the engineering

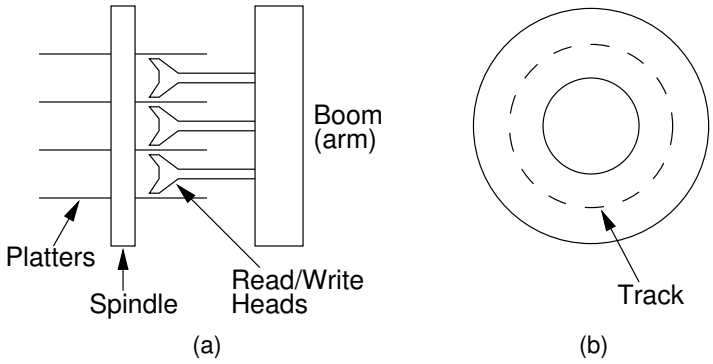

Figure 8.2 (a) A typical disk drive arranged as a stack of platters. (b) One track on a disk drive platter.

to each head. The data on a single platter that are accessible to any one position of the head for that platter are collectively called a **track**, that is, all data on a platter that are a fixed distance from the spindle, as shown in Figure 8.2(b). The collection of all tracks that are a fixed distance from the spindle is called a **cylinder**. Thus, a cylinder is all of the data that can be read when the arms are in a particular position.

Each track is subdivided into **sectors**. Between each sector there are **inter-sector gaps** in which no data are stored. These gaps allow the read head to recognize the end of a sector. Note that each sector contains the same amount of data. Because the outer tracks have greater length, they contain fewer bits per inch than do the inner tracks. Thus, about half of the potential storage space is wasted, because only the innermost tracks are stored at the highest possible data density. This arrangement is illustrated by Figure 8.3a. Disk drives today actually group tracks into "zones" such that the tracks in the innermost zone adjust their data density going out to maintain the same radial data density, then the tracks of the next zone reset the data density to make better use of their storage ability, and so on. This arrangement is shown in Figure 8.3b.

In contrast to the physical layout of a hard disk, a CD-ROM consists of a single spiral track. Bits of information along the track are equally spaced, so the information density is the same at both the outer and inner portions of the track. To keep the information flow at a constant rate along the spiral, the drive must speed up the rate of disk spin as the I/O head moves toward the center of the disk. This makes for a more complicated and slower mechanism.

Three separate steps take place when reading a particular byte or series of bytes of data from a hard disk. First, the I/O head moves so that it is positioned over the track containing the data. This movement is called a **seek**. Second, the sector containing the data rotates to come under the head. When in use the disk is always

compromises change over time. In addition, most of the description given here for disk drives is a simplified version of the reality. But this is a useful working model to understand what is going on.

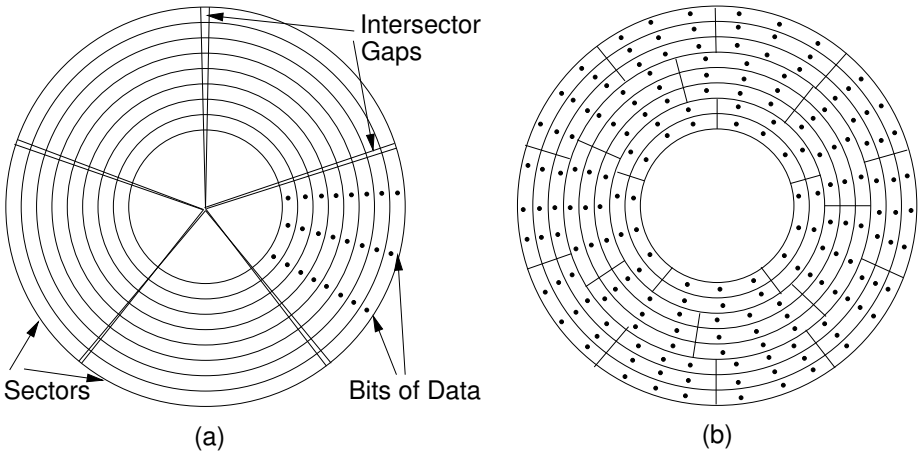

Figure 8.3 The organization of a disk platter. Dots indicate density of information. (a) Nominal arrangement of tracks showing decreasing data density when moving outward from the center of the disk. (b) A "zoned" arrangement with the sector size and density periodically reset in tracks further away from the center.

spinning. At the time of this writing, typical disk spin rates are 7200 rotations per minute (rpm). The time spent waiting for the desired sector to come under the I/O head is called **rotational delay** or **rotational latency**. The third step is the actual transfer (i.e., reading or writing) of data. It takes relatively little time to read information once the first byte is positioned under the I/O head, simply the amount of time required for it all to move under the head. In fact, disk drives are designed not to read one byte of data, but rather to read an entire sector of data at each request. Thus, a sector is the minimum amount of data that can be read or written at one time.

In general, it is desirable to keep all sectors for a file together on as few tracks as possible. This desire stems from two assumptions:

1. Seek time is slow (it is typically the most expensive part of an I/O operation), and
2. If one sector of the file is read, the next sector will probably soon be read.

Assumption (2) is called **locality of reference**, a concept that comes up frequently in computer applications.

Contiguous sectors are often grouped to form a **cluster**. A cluster is the smallest unit of allocation for a file, so all files are a multiple of the cluster size. The cluster size is determined by the operating system. The file manager keeps track of which clusters make up each file.

In Microsoft Windows systems, there is a designated portion of the disk called the **File Allocation Table**, which stores information about which sectors belong to which file. In contrast, UNIX does not use clusters. The smallest unit of file

allocation and the smallest unit that can be read/written is a sector, which in UNIX terminology is called a **block**. UNIX maintains information about file organization in certain disk blocks called **i-nodes**.

A group of physically contiguous clusters from the same file is called an **extent**. Ideally, all clusters making up a file will be contiguous on the disk (i.e., the file will consist of one extent), so as to minimize seek time required to access different portions of the file. If the disk is nearly full when a file is created, there might not be an extent available that is large enough to hold the new file. Furthermore, if a file grows, there might not be free space physically adjacent. Thus, a file might consist of several extents widely spaced on the disk. The fuller the disk, and the more that files on the disk change, the worse this file fragmentation (and the resulting seek time) becomes. File fragmentation leads to a noticeable degradation in performance as additional seeks are required to access data.

Another type of problem arises when the file's logical record size does not match the sector size. If the sector size is not a multiple of the record size (or vice versa), records will not fit evenly within a sector. For example, a sector might be 2048 bytes long, and a logical record 100 bytes. This leaves room to store 20 records with 48 bytes left over. Either the extra space is wasted, or else records are allowed to cross sector boundaries. If a record crosses a sector boundary, two disk accesses might be required to read it. If the space is left empty instead, such wasted space is called **internal fragmentation**.

A second example of internal fragmentation occurs at cluster boundaries. Files whose size is not an even multiple of the cluster size must waste some space at the end of the last cluster. The worst case will occur when file size modulo cluster size is one (for example, a file of 4097 bytes and a cluster of 4096 bytes). Thus, cluster size is a tradeoff between large files processed sequentially (where a large cluster size is desirable to minimize seeks) and small files (where small clusters are desirable to minimize wasted storage).

Every disk drive organization requires that some disk space be used to organize the sectors, clusters, and so forth. The layout of sectors within a track is illustrated by Figure 8.4. Typical information that must be stored on the disk itself includes the File Allocation Table, **sector headers** that contain address marks and information about the condition (whether usable or not) for each sector, and gaps between sectors. The sector header also contains error detection codes to help verify that the data have not been corrupted. This is why most disk drives have a "nominal" size that is greater than the actual amount of user data that can be stored on the drive. The difference is the amount of space required to organize the information on the disk. Even more space will be lost due to fragmentation.

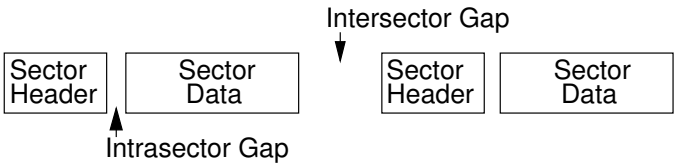

Figure 8.4 An illustration of sector gaps within a track. Each sector begins with a sector header containing the sector address and an error detection code for the contents of that sector. The sector header is followed by a small intra-sector gap, followed in turn by the sector data. Each sector is separated from the next sector by a larger inter-sector gap.

8.2.2 Disk Access Costs

When a seek is required, it is usually the primary cost when accessing information on disk. This assumes of course that a seek is necessary. When reading a file in sequential order (if the sectors comprising the file are contiguous on disk), little seeking is necessary. However, when accessing a random disk sector, seek time becomes the dominant cost for the data access. While the actual seek time is highly variable, depending on the distance between the track where the I/O head currently is and the track where the head is moving to, we will consider only two numbers. One is the track-to-track cost, or the minimum time necessary to move from a track to an adjacent track. This is appropriate when you want to analyze access times for files that are well placed on the disk. The second number is the average seek time for a random access. These two numbers are often provided by disk manufacturers. A typical example is the Western Digital Caviar serial ATA drive. The manufacturer's specifications indicate that the track-to-track time is 2.0 ms and the average seek time is 9.0 ms. In 2008 a typical drive in this line might be 120GB in size. In 2011, that same line of drives had sizes of up to 2 or 3TB. In both years, the advertised track-to-track and average seek times were identical.

For many years, typical rotation speed for disk drives was 3600 rpm, or one rotation every 16.7 ms. Most disk drives in 2011 had a rotation speed of 7200 rpm, or 8.3 ms per rotation. When reading a sector at random, you can expect that the disk will need to rotate halfway around to bring the desired sector under the I/O head, or 4.2 ms for a 7200-rpm disk drive.

Once under the I/O head, a sector of data can be transferred as fast as that sector rotates under the head. If an entire track is to be read, then it will require one rotation (8.3 ms at 7200 rpm) to move the full track under the head. If only part of the track is to be read, then proportionately less time will be required. For example, if there are 16,000 sectors on the track and one sector is to be read, this will require a trivial amount of time (1/16,000 of a rotation).

Example 8.1 Assume that an older disk drive has a total (nominal) capacity of 16.8GB spread among 10 platters, yielding 1.68GB/platter. Each

platter contains 13,085 tracks and each track contains (after formatting) 256 sectors of 512 bytes/sector. Track-to-track seek time is 2.2 ms and average seek time for random access is 9.5 ms. Assume the operating system maintains a cluster size of 8 sectors per cluster (4KB), yielding 32 clusters per track. The disk rotation rate is 5400 rpm (11.1 ms per rotation). Based on this information we can estimate the cost for various file processing operations.

How much time is required to read the track? On average, it will require half a rotation to bring the first sector of the track under the I/O head, and then one complete rotation to read the track.

How long will it take to read a file of 1MB divided into 2048 sector-sized (512 byte) records? This file will be stored in 256 clusters, because each cluster holds 8 sectors. The answer to the question depends largely on how the file is stored on the disk, that is, whether it is all together or broken into multiple extents. We will calculate both cases to see how much difference this makes.

If the file is stored so as to fill all of the sectors of eight adjacent tracks, then the cost to read the first sector will be the time to seek to the first track (assuming this requires a random seek), then a wait for the initial rotational delay, and then the time to read (which is the same as the time to rotate the disk again). This requires

$$9.5 + 11.1 \times 1.5 = 26.2 \text{ ms}.$$

At this point, because we assume that the next seven tracks require only a track-to-track seek because they are adjacent. Each requires

$$2.2 + 11.1 \times 1.5 = 18.9 \text{ ms}.$$

The total time required is therefore

$$26.2\text{ms} + 7 \times 18.9\text{ms} = 158.5\text{ms}.$$

If the file's clusters are spread randomly across the disk, then we must perform a seek for each cluster, followed by the time for rotational delay. Once the first sector of the cluster comes under the I/O head, very little time is needed to read the cluster because only 8/256 of the track needs to rotate under the head, for a total time of about 5.9 ms for latency and read time. Thus, the total time required is about

$$256(9.5 + 5.9) \approx 3942\text{ms}$$

or close to 4 seconds. This is much longer than the time required when the file is all together on disk!

This example illustrates why it is important to keep disk files from becoming fragmented, and why so-called "disk defragmenters" can speed up file processing time. File fragmentation happens most commonly when the disk is nearly full and the file manager must search for free space whenever a file is created or changed.

8.3 Buffers and Buffer Pools

Given the specifications of the disk drive from Example 8.1, we find that it takes about $9.5+11.1\times1.5 = 26.2$ ms to read one track of data on average. It takes about $9.5+11.1/2+(1/256)\times11.1 = 15.1$ ms on average to read a single sector of data. This is a good savings (slightly over half the time), but less than 1% of the data on the track are read. If we want to read only a single byte, it would save us effectively no time over that required to read an entire sector. For this reason, nearly all disk drives automatically read or write an entire sector's worth of information whenever the disk is accessed, even when only one byte of information is requested.

Once a sector is read, its information is stored in main memory. This is known as **buffering** or **caching** the information. If the next disk request is to that same sector, then it is not necessary to read from disk again because the information is already stored in main memory. Buffering is an example of one method for minimizing disk accesses mentioned at the beginning of the chapter: Bring off additional information from disk to satisfy future requests. If information from files were accessed at random, then the chance that two consecutive disk requests are to the same sector would be low. However, in practice most disk requests are close to the location (in the logical file at least) of the previous request. This means that the probability of the next request "hitting the cache" is much higher than chance would indicate.

This principle explains one reason why average access times for new disk drives are lower than in the past. Not only is the hardware faster, but information is also now stored using better algorithms and larger caches that minimize the number of times information needs to be fetched from disk. This same concept is also used to store parts of programs in faster memory within the CPU, using the CPU cache that is prevalent in modern microprocessors.

Sector-level buffering is normally provided by the operating system and is often built directly into the disk drive controller hardware. Most operating systems maintain at least two buffers, one for input and one for output. Consider what would happen if there were only one buffer during a byte-by-byte copy operation. The sector containing the first byte would be read into the I/O buffer. The output operation would need to destroy the contents of the single I/O buffer to write this byte. Then the buffer would need to be filled again from disk for the second byte,

only to be destroyed during output. The simple solution to this problem is to keep one buffer for input, and a second for output.

Most disk drive controllers operate independently from the CPU once an I/O request is received. This is useful because the CPU can typically execute millions of instructions during the time required for a single I/O operation. A technique that takes maximum advantage of this micro-parallelism is **double buffering**. Imagine that a file is being processed sequentially. While the first sector is being read, the CPU cannot process that information and so must wait or find something else to do in the meantime. Once the first sector is read, the CPU can start processing while the disk drive (in parallel) begins reading the second sector. If the time required for the CPU to process a sector is approximately the same as the time required by the disk controller to read a sector, it might be possible to keep the CPU continuously fed with data from the file. The same concept can also be applied to output, writing one sector to disk while the CPU is writing to a second output buffer in memory. Thus, in computers that support double buffering, it pays to have at least two input buffers and two output buffers available.

Caching information in memory is such a good idea that it is usually extended to multiple buffers. The operating system or an application program might store many buffers of information taken from some **backing storage** such as a disk file. This process of using buffers as an intermediary between a user and a disk file is called **buffering** the file. The information stored in a buffer is often called a **page**, and the collection of buffers is called a **buffer pool**. The goal of the buffer pool is to increase the amount of information stored in memory in hopes of increasing the likelihood that new information requests can be satisfied from the buffer pool rather than requiring new information to be read from disk.

As long as there is an unused buffer available in the buffer pool, new information can be read in from disk on demand. When an application continues to read new information from disk, eventually all of the buffers in the buffer pool will become full. Once this happens, some decision must be made about what information in the buffer pool will be sacrificed to make room for newly requested information.

When replacing information contained in the buffer pool, the goal is to select a buffer that has "unnecessary" information, that is, the information least likely to be requested again. Because the buffer pool cannot know for certain what the pattern of future requests will look like, a decision based on some **heuristic**, or best guess, must be used. There are several approaches to making this decision.

One heuristic is "first-in, first-out" (FIFO). This scheme simply orders the buffers in a queue. The buffer at the front of the queue is used next to store new information and then placed at the end of the queue. In this way, the buffer to be replaced is the one that has held its information the longest, in hopes that this information is no longer needed. This is a reasonable assumption when processing moves along the file at some steady pace in roughly sequential order. However,

many programs work with certain key pieces of information over and over again, and the importance of information has little to do with how long ago the information was first accessed. Typically it is more important to know how many times the information has been accessed, or how recently the information was last accessed.

Another approach is called "least frequently used" (LFU). LFU tracks the number of accesses to each buffer in the buffer pool. When a buffer must be reused, the buffer that has been accessed the fewest number of times is considered to contain the "least important" information, and so it is used next. LFU, while it seems intuitively reasonable, has many drawbacks. First, it is necessary to store and update access counts for each buffer. Second, what was referenced many times in the past might now be irrelevant. Thus, some time mechanism where counts "expire" is often desirable. This also avoids the problem of buffers that slowly build up big counts because they get used just often enough to avoid being replaced. An alternative is to maintain counts for all sectors ever read, not just the sectors currently in the buffer pool. This avoids immediately replacing the buffer just read, which has not yet had time to build a high access count.

The third approach is called "least recently used" (LRU). LRU simply keeps the buffers in a list. Whenever information in a buffer is accessed, this buffer is brought to the front of the list. When new information must be read, the buffer at the back of the list (the one least recently used) is taken and its "old" information is either discarded or written to disk, as appropriate. This is an easily implemented approximation to LFU and is often the method of choice for managing buffer pools unless special knowledge about information access patterns for an application suggests a special-purpose buffer management scheme.

The main purpose of a buffer pool is to minimize disk I/O. When the contents of a block are modified, we could write the updated information to disk immediately. But what if the block is changed again? If we write the block's contents after every change, that might be a lot of disk write operations that can be avoided. It is more efficient to wait until either the file is to be closed, or the contents of the buffer containing that block is to be flushed from the buffer pool.

When a buffer's contents are to be replaced in the buffer pool, we only want to write the contents to disk if it is necessary. That would be necessary only if the contents have changed since the block was read in originally from the file. The way to insure that the block is written when necessary, but only when necessary, is to maintain a Boolean variable with the buffer (often referred to as the **dirty bit**) that is turned on when the buffer's contents are modified by the client. At the time when the block is flushed from the buffer pool, it is written to disk if and only if the dirty bit has been turned on.

Modern operating systems support **virtual memory**. Virtual memory is a technique that allows the programmer to write programs as though there is more of the faster main memory (such as RAM) than actually exists. Virtual memory makes use

of a buffer pool to store data read from blocks on slower, secondary memory (such as on the disk drive). The disk stores the complete contents of the virtual memory. Blocks are read into main memory as demanded by memory accesses. Naturally, programs using virtual memory techniques are slower than programs whose data are stored completely in main memory. The advantage is reduced programmer effort because a good virtual memory system provides the appearance of larger main memory without modifying the program.

Example 8.2 Consider a virtual memory whose size is ten sectors, and which has a buffer pool of five buffers (each one sector in size) associated with it. We will use a LRU replacement scheme. The following series of memory requests occurs.

$$9017668135171$$

After the first five requests, the buffer pool will store the sectors in the order 6, 7, 1, 0, 9. Because Sector 6 is already at the front, the next request can be answered without reading new data from disk or reordering the buffers. The request to Sector 8 requires emptying the contents of the least recently used buffer, which contains Sector 9. The request to Sector 1 brings the buffer holding Sector 1's contents back to the front. Processing the remaining requests results in the buffer pool as shown in Figure 8.5.

Example 8.3 Figure 8.5 illustrates a buffer pool of five blocks mediating a virtual memory of ten blocks. At any given moment, up to five sectors of information can be in main memory. Assume that Sectors 1, 7, 5, 3, and 8 are currently in the buffer pool, stored in this order, and that we use the LRU buffer replacement strategy. If a request for Sector 9 is then received, then one sector currently in the buffer pool must be replaced. Because the buffer containing Sector 8 is the least recently used buffer, its contents will be copied back to disk at Sector 8. The contents of Sector 9 are then copied into this buffer, and it is moved to the front of the buffer pool (leaving the buffer containing Sector 3 as the new least-recently used buffer). If the next memory request were to Sector 5, no data would need to be read from disk. Instead, the buffer already containing Sector 5 would be moved to the front of the buffer pool.

When implementing buffer pools, there are two basic approaches that can be taken regarding the transfer of information between the user of the buffer pool and the buffer pool class itself. The first approach is to pass "messages" between the two. This approach is illustrated by the following abstract class:

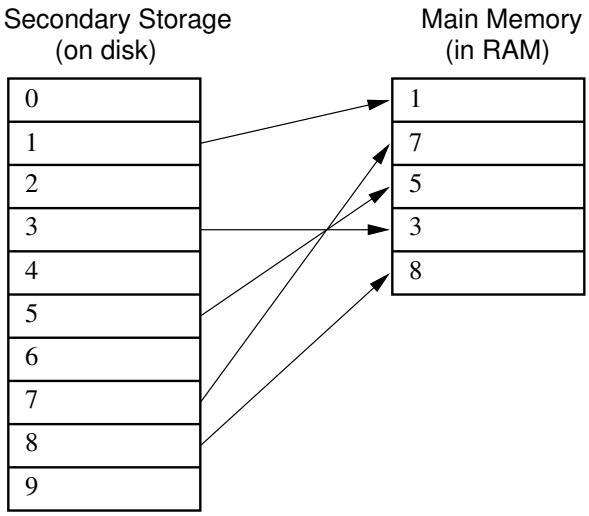

Figure 8.5 An illustration of virtual memory. The complete collection of information resides in the slower, secondary storage (on disk). Those sectors recently accessed are held in the fast main memory (in RAM). In this example, copies of Sectors 1, 7, 5, 3, and 8 from secondary storage are currently stored in the main memory. If a memory access to Sector 9 is received, one of the sectors currently in main memory must be replaced.

```
/** ADT for buffer pools using the message-passing style */
public interface BufferPoolADT {
  /** Copy "sz" bytes from "space" to position "pos" in the
      buffered storage */
  public void insert(byte[] space, int sz, int pos);

  /** Copy "sz" bytes from position "pos" of the buffered
      storage to "space". */
  public void getbytes(byte[] space, int sz, int pos);
}
```

This simple class provides an interface with two member functions, **`insert`** and **`getbytes`**. The information is passed between the buffer pool user and the buffer pool through the **`space`** parameter. This is storage space, provided by the bufferpool client and at least **`sz`** bytes long, which the buffer pool can take information from (the **`insert`** function) or put information into (the **`getbytes`** function). Parameter **`pos`** indicates where the information will be placed in the buffer pool's logical storage space. Physically, it will actually be copied to the appropriate byte position in some buffer in the buffer pool. This ADT is similar to the **`read`** and **`write`** methods of the **`RandomAccessFile`** class discussed in Section 8.4.

Example 8.4 Assume each sector of the disk file (and thus each block in the buffer pool) stores 1024 bytes. Assume that the buffer pool is in the state shown in Figure 8.5. If the next request is to copy 40 bytes beginning at position 6000 of the file, these bytes should be placed into Sector 5 (whose bytes go from position 5120 to position 6143). Because Sector 5 is currently in the buffer pool, we simply copy the 40 bytes contained in **space** to byte positions 880-919. The buffer containing Sector 5 is then moved to the buffer pool ahead of the buffer containing Sector 1.

An alternative interface is to have the buffer pool provide to the user a direct pointer to a buffer that contains the requested information. Such an interface might look as follows:

```
/** ADT for buffer pools using the buffer-passing style */
public interface BufferPoolADT {
  /** Return pointer to the requested block */
  public byte[] getblock(int block);

  /** Set the dirty bit for the buffer holding "block" */
  public void dirtyblock(int block);

  // Tell the size of a buffer
  public int blocksize();
};
```

In this approach, the buffer pool user is made aware that the storage space is divided into blocks of a given size, where each block is the size of a buffer. The user requests specific blocks from the buffer pool, with a pointer to the buffer holding the requested block being returned to the user. The user might then read from or write to this space. If the user writes to the space, the buffer pool must be informed of this fact. The reason is that, when a given block is to be removed from the buffer pool, the contents of that block must be written to the backing storage if it has been modified. If the block has not been modified, then it is unnecessary to write it out.

Example 8.5 We wish to write 40 bytes beginning at logical position 6000 in the file. Assume that the buffer pool is in the state shown in Figure 8.5. Using the second ADT, the client would need to know that blocks (buffers) are of size 1024, and therefore would request access to Sector 5. A pointer to the buffer containing Sector 5 would be returned by the call to **getblock**. The client would then copy 40 bytes to positions 880-919 of the buffer, and call **dirtyblock** to warn the buffer pool that the contents of this block have been modified.

A variation on this approach is to have the **`getblock`** function take another parameter to indicate the "mode" of use for the information. If the mode is READ then the buffer pool assumes that no changes will be made to the buffer's contents (and so no write operation need be done when the buffer is reused to store another block). If the mode is WRITE then the buffer pool assumes that the client will not look at the contents of the buffer and so no read from the file is necessary. If the mode is READ AND WRITE then the buffer pool would read the existing contents of the block in from disk, and write the contents of the buffer to disk when the buffer is to be reused. Using the "mode" approach, the **`dirtyblock`** method is avoided.

One problem with the buffer-passing ADT is the risk of **stale pointers**. When the buffer pool user is given a pointer to some buffer space at time **T1**, that pointer does indeed refer to the desired data at that time. As further requests are made to the buffer pool, it is possible that the data in any given buffer will be removed and replaced with new data. If the buffer pool user at a later time **T2** then refers to the data referred to by the pointer given at time **T1**, it is possible that the data are no longer valid because the buffer contents have been replaced in the meantime. Thus the pointer into the buffer pool's memory has become "stale." To guarantee that a pointer is not stale, it should not be used if intervening requests to the buffer pool have taken place.

We can solve this problem by introducing the concept of a user (or possibly multiple users) gaining access to a buffer, and then releasing the buffer when done. We will add method **`acquireBuffer`** and **`releaseBuffer`** for this purpose. Method **`acquireBuffer`** takes a block ID as input and returns a pointer to the buffer that will be used to store this block. The buffer pool will keep a count of the number of requests currently active for this block. Method **`releaseBuffer`** will reduce the count of active users for the associated block. Buffers associated with active blocks will not be eligible for flushing from the buffer pool. This will lead to a problem if the client neglects to release active blocks when they are no longer needed. There would also be a problem if there were more total active blocks than buffers in the buffer pool. However, the buffer pool should always be initialized to include more buffers than should ever be active at one time.

An additional problem with both ADTs presented so far comes when the user intends to completely overwrite the contents of a block, and does not need to read in the old contents already on disk. However, the buffer pool cannot in general know whether the user wishes to use the old contents or not. This is especially true with the message-passing approach where a given message might overwrite only part of the block. In this case, the block will be read into memory even when not needed, and then its contents will be overwritten.

This inefficiency can be avoided (at least in the buffer-passing version) by separating the assignment of blocks to buffers from actually reading in data for the

block. In particular, the following revised buffer-passing ADT does not actually read data in the **acquireBuffer** method. Users who wish to see the old contents must then issue a **readBlock** request to read the data from disk into the buffer, and then a **getDataPointer** request to gain direct access to the buffer's data contents.

```
/** Improved ADT for buffer pools using the buffer-passing
    style. Most user functionality is in the buffer class,
    not the buffer pool itself. */

/** A single buffer in the buffer pool */
public interface BufferADT {
  /** Read the associated block from disk (if necessary)
      and return a pointer to the data */
  public byte[] readBlock();

  /** Return a pointer to the buffer's data array
      (without reading from disk) */
  public byte[] getDataPointer();

  /** Flag buffer's contents as having changed, so that
      flushing the block will write it back to disk */
  public void markDirty();

  /** Release the block's access to this buffer. Further
      accesses to this buffer are illegal. */
  public void releaseBuffer();
}

/** The bufferpool */
public interface BufferPoolADT {

  /** Relate a block to a buffer, returning a pointer to
      a buffer object */
  Buffer acquireBuffer(int block);
}
```

Again, a mode parameter could be added to the **acquireBuffer** method, eliminating the need for the **readBlock** and **markDirty** methods.

Clearly, the buffer-passing approach places more obligations on the user of the buffer pool. These obligations include knowing the size of a block, not corrupting the buffer pool's storage space, and informing the buffer pool both when a block has been modified and when it is no longer needed. So many obligations make this approach prone to error. An advantage is that there is no need to do an extra copy step when getting information from the user to the buffer. If the size of the records stored is small, this is not an important consideration. If the size of the records is large (especially if the record size and the buffer size are the same, as typically is the case when implementing B-trees, see Section 10.5), then this efficiency issue might

become important. Note however that the in-memory copy time will always be far less than the time required to write the contents of a buffer to disk. For applications where disk I/O is the bottleneck for the program, even the time to copy lots of information between the buffer pool user and the buffer might be inconsequential. Another advantage to buffer passing is the reduction in unnecessary read operations for data that will be overwritten anyway.

You should note that these implementations for the buffer pool ADT do not use generics. Instead, the **`space`** parameter and the buffer pointer are declared to be **`byte[]`**. When a class uses a generic, that means that the record type is arbitrary, but that the class knows what the record type is. In contrast, using **`byte[]`** for the space means that not only is the record type arbitrary, but also the buffer pool does not even know what the user's record type is. In fact, a given buffer pool might have many users who store many types of records.

In a buffer pool, the user decides where a given record will be stored but has no control over the precise mechanism by which data are transferred to the backing storage. This is in contrast to the memory manager described in Section 12.3 in which the user passes a record to the manager and has no control at all over where the record is stored.

8.4 The Programmer's View of Files

The Java programmer's logical view of a random access file is a single stream of bytes. Interaction with a file can be viewed as a communications channel for issuing one of three instructions: read bytes from the current position in the file, write bytes to the current position in the file, and move the current position within the file. You do not normally see how the bytes are stored in sectors, clusters, and so forth. The mapping from logical to physical addresses is done by the file system, and sector-level buffering is done automatically by the disk controller.

When processing records in a disk file, the order of access can have a great effect on I/O time. A **random access** procedure processes records in an order independent of their logical order within the file. **Sequential access** processes records in order of their logical appearance within the file. Sequential processing requires less seek time if the physical layout of the disk file matches its logical layout, as would be expected if the file were created on a disk with a high percentage of free space.

Java provides several mechanisms for manipulating disk files. One of the most commonly used is the **`RandomAccessFile`** class. The following methods can be used to manipulate information in the file.

- **`RandomAccessFile(String name, String mode)`**: Class constructor, opens a disk file for processing.

- **`read(byte[] b)`**: Read some bytes from the current position in the file. The current position moves forward as the bytes are read.
- **`write(byte[] b)`**: Write some bytes at the current position in the file (overwriting the bytes already at that position). The current position moves forward as the bytes are written.
- **`seek(long pos)`**: Move the current position in the file to **`pos`**. This allows bytes at arbitrary places within the file to be read or written.
- **`close()`**: Close a file at the end of processing.

8.5 External Sorting

We now consider the problem of sorting collections of records too large to fit in main memory. Because the records must reside in peripheral or external memory, such sorting methods are called **external sorts**. This is in contrast to the internal sorts discussed in Chapter 7 which assume that the records to be sorted are stored in main memory. Sorting large collections of records is central to many applications, such as processing payrolls and other large business databases. As a consequence, many external sorting algorithms have been devised. Years ago, sorting algorithm designers sought to optimize the use of specific hardware configurations, such as multiple tape or disk drives. Most computing today is done on personal computers and low-end workstations with relatively powerful CPUs, but only one or at most two disk drives. The techniques presented here are geared toward optimized processing on a single disk drive. This approach allows us to cover the most important issues in external sorting while skipping many less important machine-dependent details. Readers who have a need to implement efficient external sorting algorithms that take advantage of more sophisticated hardware configurations should consult the references in Section 8.6.

When a collection of records is too large to fit in main memory, the only practical way to sort it is to read some records from disk, do some rearranging, then write them back to disk. This process is repeated until the file is sorted, with each record read perhaps many times. Given the high cost of disk I/O, it should come as no surprise that the primary goal of an external sorting algorithm is to minimize the number of times information must be read from or written to disk. A certain amount of additional CPU processing can profitably be traded for reduced disk access.

Before discussing external sorting techniques, consider again the basic model for accessing information from disk. The file to be sorted is viewed by the programmer as a sequential series of fixed-size **blocks**. Assume (for simplicity) that each block contains the same number of fixed-size data records. Depending on the application, a record might be only a few bytes — composed of little or nothing more than the key — or might be hundreds of bytes with a relatively small key field. Records are assumed not to cross block boundaries. These assumptions can be

relaxed for special-purpose sorting applications, but ignoring such complications makes the principles clearer.

As explained in Section 8.2, a sector is the basic unit of I/O. In other words, all disk reads and writes are for one or more complete sectors. Sector sizes are typically a power of two, in the range 512 to 16K bytes, depending on the operating system and the size and speed of the disk drive. The block size used for external sorting algorithms should be equal to or a multiple of the sector size.

Under this model, a sorting algorithm reads a block of data into a buffer in main memory, performs some processing on it, and at some future time writes it back to disk. From Section 8.1 we see that reading or writing a block from disk takes on the order of one million times longer than a memory access. Based on this fact, we can reasonably expect that the records contained in a single block can be sorted by an internal sorting algorithm such as Quicksort in less time than is required to read or write the block.

Under good conditions, reading from a file in sequential order is more efficient than reading blocks in random order. Given the significant impact of seek time on disk access, it might seem obvious that sequential processing is faster. However, it is important to understand precisely under what circumstances sequential file processing is actually faster than random access, because it affects our approach to designing an external sorting algorithm.

Efficient sequential access relies on seek time being kept to a minimum. The first requirement is that the blocks making up a file are in fact stored on disk in sequential order and close together, preferably filling a small number of contiguous tracks. At the very least, the number of extents making up the file should be small. Users typically do not have much control over the layout of their file on disk, but writing a file all at once in sequential order to a disk drive with a high percentage of free space increases the likelihood of such an arrangement.

The second requirement is that the disk drive's I/O head remain positioned over the file throughout sequential processing. This will not happen if there is competition of any kind for the I/O head. For example, on a multi-user time-shared computer the sorting process might compete for the I/O head with the processes of other users. Even when the sorting process has sole control of the I/O head, it is still likely that sequential processing will not be efficient. Imagine the situation where all processing is done on a single disk drive, with the typical arrangement of a single bank of read/write heads that move together over a stack of platters. If the sorting process involves reading from an input file, alternated with writing to an output file, then the I/O head will continuously seek between the input file and the output file. Similarly, if two input files are being processed simultaneously (such as during a merge process), then the I/O head will continuously seek between these two files.

The moral is that, with a single disk drive, there often is no such thing as efficient sequential processing of a data file. Thus, a sorting algorithm might be more efficient if it performs a smaller number of non-sequential disk operations rather than a larger number of logically sequential disk operations that require a large number of seeks in practice.

As mentioned previously, the record size might be quite large compared to the size of the key. For example, payroll entries for a large business might each store hundreds of bytes of information including the name, ID, address, and job title for each employee. The sort key might be the ID number, requiring only a few bytes. The simplest sorting algorithm might be to process such records as a whole, reading the entire record whenever it is processed. However, this will greatly increase the amount of I/O required, because only a relatively few records will fit into a single disk block. Another alternative is to do a **key sort**. Under this method, the keys are all read and stored together in an **index file**, where each key is stored along with a pointer indicating the position of the corresponding record in the original data file. The key and pointer combination should be substantially smaller than the size of the original record; thus, the index file will be much smaller than the complete data file. The index file will then be sorted, requiring much less I/O because the index records are smaller than the complete records.

Once the index file is sorted, it is possible to reorder the records in the original database file. This is typically not done for two reasons. First, reading the records in sorted order from the record file requires a random access for each record. This can take a substantial amount of time and is only of value if the complete collection of records needs to be viewed or processed in sorted order (as opposed to a search for selected records). Second, database systems typically allow searches to be done on multiple keys. For example, today's processing might be done in order of ID numbers. Tomorrow, the boss might want information sorted by salary. Thus, there might be no single "sorted" order for the full record. Instead, multiple index files are often maintained, one for each sort key. These ideas are explored further in Chapter 10.

8.5.1 Simple Approaches to External Sorting

If your operating system supports virtual memory, the simplest "external" sort is to read the entire file into virtual memory and run an internal sorting method such as Quicksort. This approach allows the virtual memory manager to use its normal buffer pool mechanism to control disk accesses. Unfortunately, this might not always be a viable option. One potential drawback is that the size of virtual memory is usually limited to something much smaller than the disk space available. Thus, your input file might not fit into virtual memory. Limited virtual memory can be overcome by adapting an internal sorting method to make use of your own buffer pool.

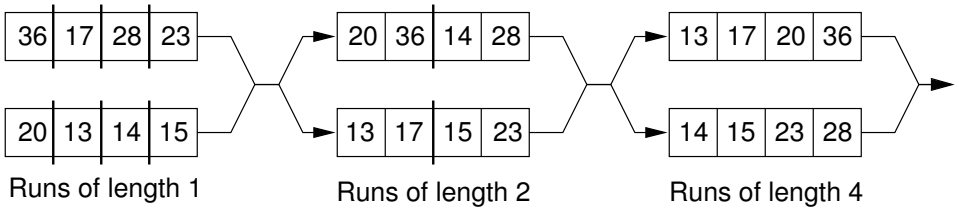

Figure 8.6 A simple external Mergesort algorithm. Input records are divided equally between two input files. The first runs from each input file are merged and placed into the first output file. The second runs from each input file are merged and placed in the second output file. Merging alternates between the two output files until the input files are empty. The roles of input and output files are then reversed, allowing the runlength to be doubled with each pass.

A more general problem with adapting an internal sorting algorithm to external sorting is that it is not likely to be as efficient as designing a new algorithm with the specific goal of minimizing disk I/O. Consider the simple adaptation of Quicksort to use a buffer pool. Quicksort begins by processing the entire array of records, with the first partition step moving indices inward from the two ends. This can be implemented efficiently using a buffer pool. However, the next step is to process each of the subarrays, followed by processing of sub-subarrays, and so on. As the subarrays get smaller, processing quickly approaches random access to the disk drive. Even with maximum use of the buffer pool, Quicksort still must read and write each record $\log n$ times on average. We can do much better. Finally, even if the virtual memory manager can give good performance using a standard Quicksort, this will come at the cost of using a lot of the system's working memory, which will mean that the system cannot use this space for other work. Better methods can save time while also using less memory.

Our approach to external sorting is derived from the Mergesort algorithm. The simplest form of external Mergesort performs a series of sequential passes over the records, merging larger and larger sublists on each pass. The first pass merges sublists of size 1 into sublists of size 2; the second pass merges the sublists of size 2 into sublists of size 4; and so on. A sorted sublist is called a **run**. Thus, each pass is merging pairs of runs to form longer runs. Each pass copies the contents of the file to another file. Here is a sketch of the algorithm, as illustrated by Figure 8.6.

1. Split the original file into two equal-sized **run files**.
2. Read one block from each run file into input buffers.
3. Take the first record from each input buffer, and write a run of length two to an output buffer in sorted order.
4. Take the next record from each input buffer, and write a run of length two to a second output buffer in sorted order.
5. Repeat until finished, alternating output between the two output run buffers. Whenever the end of an input block is reached, read the next block from the

appropriate input file. When an output buffer is full, write it to the appropriate output file.

6. Repeat steps 2 through 5, using the original output files as input files. On the second pass, the first two records of each input run file are already in sorted order. Thus, these two runs may be merged and output as a single run of four elements.
7. Each pass through the run files provides larger and larger runs until only one run remains.

Example 8.6 Using the input of Figure 8.6, we first create runs of length one split between two input files. We then process these two input files sequentially, making runs of length two. The first run has the values 20 and 36, which are output to the first output file. The next run has 13 and 17, which is output to the second file. The run 14, 28 is sent to the first file, then run 15, 23 is sent to the second file, and so on. Once this pass has completed, the roles of the input files and output files are reversed. The next pass will merge runs of length two into runs of length four. Runs 20, 36 and 13, 17 are merged to send 13, 17, 20, 36 to the first output file. Then runs 14, 28 and 15, 23 are merged to send run 14, 15, 23, 28 to the second output file. In the final pass, these runs are merged to form the final run 13, 14, 15, 17, 20, 23, 28, 36.

This algorithm can easily take advantage of the double buffering techniques described in Section 8.3. Note that the various passes read the input run files sequentially and write the output run files sequentially. For sequential processing and double buffering to be effective, however, it is necessary that there be a separate I/O head available for each file. This typically means that each of the input and output files must be on separate disk drives, requiring a total of four disk drives for maximum efficiency.

The external Mergesort algorithm just described requires that $\log n$ passes be made to sort a file of n records. Thus, each record must be read from disk and written to disk $\log n$ times. The number of passes can be significantly reduced by observing that it is not necessary to use Mergesort on small runs. A simple modification is to read in a block of data, sort it in memory (perhaps using Quicksort), and then output it as a single sorted run.

Example 8.7 Assume that we have blocks of size 4KB, and records are eight bytes with four bytes of data and a 4-byte key. Thus, each block contains 512 records. Standard Mergesort would require nine passes to generate runs of 512 records, whereas processing each block as a unit can be done

in one pass with an internal sort. These runs can then be merged by Mergesort. Standard Mergesort requires eighteen passes to process 256K records. Using an internal sort to create initial runs of 512 records reduces this to one initial pass to create the runs and nine merge passes to put them all together, approximately half as many passes.

We can extend this concept to improve performance even further. Available main memory is usually much more than one block in size. If we process larger initial runs, then the number of passes required by Mergesort is further reduced. For example, most modern computers can provide tens or even hundreds of megabytes of RAM to the sorting program. If all of this memory (excepting a small amount for buffers and local variables) is devoted to building initial runs as large as possible, then quite large files can be processed in few passes. The next section presents a technique for producing large runs, typically twice as large as could fit directly into main memory.

Another way to reduce the number of passes required is to increase the number of runs that are merged together during each pass. While the standard Mergesort algorithm merges two runs at a time, there is no reason why merging needs to be limited in this way. Section 8.5.3 discusses the technique of multiway merging.

Over the years, many variants on external sorting have been presented, but all are based on the following two steps:

1. Break the file into large initial runs.
2. Merge the runs together to form a single sorted file.

8.5.2 Replacement Selection

This section treats the problem of creating initial runs as large as possible from a disk file, assuming a fixed amount of RAM is available for processing. As mentioned previously, a simple approach is to allocate as much RAM as possible to a large array, fill this array from disk, and sort the array using Quicksort. Thus, if the size of memory available for the array is M records, then the input file can be broken into initial runs of length M. A better approach is to use an algorithm called **replacement selection** that, on average, creates runs of $2M$ records in length. Replacement selection is actually a slight variation on the Heapsort algorithm. The fact that Heapsort is slower than Quicksort is irrelevant in this context because I/O time will dominate the total running time of any reasonable external sorting algorithm. Building longer initial runs will reduce the total I/O time required.

Replacement selection views RAM as consisting of an array of size M in addition to an input buffer and an output buffer. (Additional I/O buffers might be desirable if the operating system supports double buffering, because replacement selection does sequential processing on both its input and its output.) Imagine that

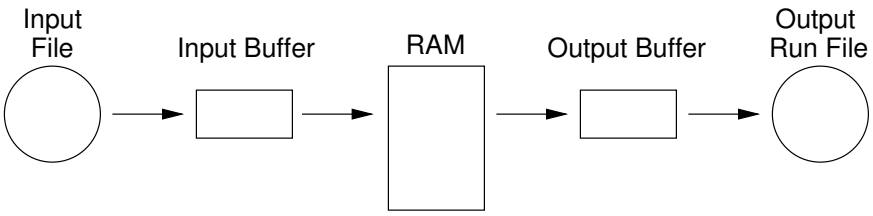

Figure 8.7 Overview of replacement selection. Input records are processed sequentially. Initially RAM is filled with M records. As records are processed, they are written to an output buffer. When this buffer becomes full, it is written to disk. Meanwhile, as replacement selection needs records, it reads them from the input buffer. Whenever this buffer becomes empty, the next block of records is read from disk.

the input and output files are streams of records. Replacement selection takes the next record in sequential order from the input stream when needed, and outputs runs one record at a time to the output stream. Buffering is used so that disk I/O is performed one block at a time. A block of records is initially read and held in the input buffer. Replacement selection removes records from the input buffer one at a time until the buffer is empty. At this point the next block of records is read in. Output to a buffer is similar: Once the buffer fills up it is written to disk as a unit. This process is illustrated by Figure 8.7.

Replacement selection works as follows. Assume that the main processing is done in an array of size M records.

1. Fill the array from disk. Set LAST $= M - 1$.
2. Build a min-heap. (Recall that a min-heap is defined such that the record at each node has a key value *less* than the key values of its children.)
3. Repeat until the array is empty:
   - **(a)** Send the record with the minimum key value (the root) to the output buffer.
   - **(b)** Let *R* be the next record in the input buffer. If *R*'s key value is greater than the key value just output ...
     - **i.** Then place *R* at the root.
     - **ii.** Else replace the root with the record in array position LAST, and place *R* at position LAST. Set LAST $=$ LAST $- 1$.
   - **(c)** Sift down the root to reorder the heap.

When the test at step 3(b) is successful, a new record is added to the heap, eventually to be output as part of the run. As long as records coming from the input file have key values greater than the last key value output to the run, they can be safely added to the heap. Records with smaller key values cannot be output as part of the current run because they would not be in sorted order. Such values must be

stored somewhere for future processing as part of another run. However, because the heap will shrink by one element in this case, there is now a free space where the last element of the heap used to be! Thus, replacement selection will slowly shrink the heap and at the same time use the discarded heap space to store records for the next run. Once the first run is complete (i.e., the heap becomes empty), the array will be filled with records ready to be processed for the second run. Figure 8.8 illustrates part of a run being created by replacement selection.

It should be clear that the minimum length of a run will be M records if the size of the heap is M, because at least those records originally in the heap will be part of the run. Under good conditions (e.g., if the input is sorted), then an arbitrarily long run is possible. In fact, the entire file could be processed as one run. If conditions are bad (e.g., if the input is reverse sorted), then runs of only size M result.

What is the expected length of a run generated by replacement selection? It can be deduced from an analogy called the **snowplow argument**. Imagine that a snowplow is going around a circular track during a heavy, but steady, snowstorm. After the plow has been around at least once, snow on the track must be as follows. Immediately behind the plow, the track is empty because it was just plowed. The greatest level of snow on the track is immediately in front of the plow, because this is the place least recently plowed. At any instant, there is a certain amount of snow S on the track. Snow is constantly falling throughout the track at a steady rate, with some snow falling "in front" of the plow and some "behind" the plow. (On a circular track, everything is actually "in front" of the plow, but Figure 8.9 illustrates the idea.) During the next revolution of the plow, all snow S on the track is removed, plus half of what falls. Because everything is assumed to be in steady state, after one revolution S snow is still on the track, so $2S$ snow must fall during a revolution, and $2S$ snow is removed during a revolution (leaving S snow behind).

At the beginning of replacement selection, nearly all values coming from the input file are greater (i.e., "in front of the plow") than the latest key value output for this run, because the run's initial key values should be small. As the run progresses, the latest key value output becomes greater and so new key values coming from the input file are more likely to be too small (i.e., "after the plow"); such records go to the bottom of the array. The total length of the run is expected to be twice the size of the array. Of course, this assumes that incoming key values are evenly distributed within the key range (in terms of the snowplow analogy, we assume that snow falls evenly throughout the track). Sorted and reverse sorted inputs do not meet this expectation and so change the length of the run.

8.5.3 Multiway Merging

The second stage of a typical external sorting algorithm merges the runs created by the first stage. Assume that we have R runs to merge. If a simple two-way merge is used, then R runs (regardless of their sizes) will require $\log R$ passes through

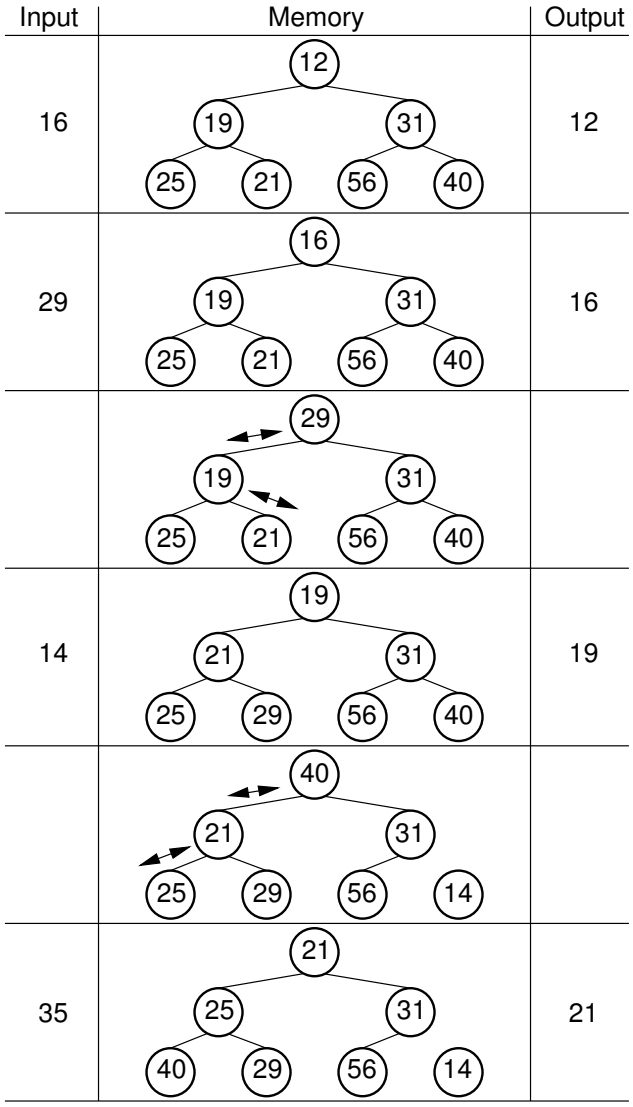

Figure 8.8 Replacement selection example. After building the heap, root value 12 is output and incoming value 16 replaces it. Value 16 is output next, replaced with incoming value 29. The heap is reordered, with 19 rising to the root. Value 19 is output next. Incoming value 14 is too small for this run and is placed at end of the array, moving value 40 to the root. Reordering the heap results in 21 rising to the root, which is output next.

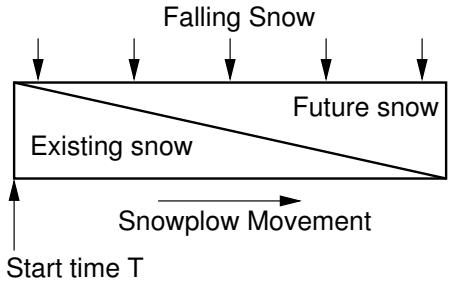

Figure 8.9 The snowplow analogy showing the action during one revolution of the snowplow. A circular track is laid out straight for purposes of illustration, and is shown in cross section. At any time T, the most snow is directly in front of the snowplow. As the plow moves around the track, the same amount of snow is always in front of the plow. As the plow moves forward, less of this is snow that was in the track at time T; more is snow that has fallen since.

the file. While R should be much less than the total number of records (because the initial runs should each contain many records), we would like to reduce still further the number of passes required to merge the runs together. Note that two-way merging does not make good use of available memory. Because merging is a sequential process on the two runs, only one block of records per run need be in memory at a time. Keeping more than one block of a run in memory at any time will not reduce the disk I/O required by the merge process (though if several blocks are read from a file at once time, at least they take advantage of sequential access). Thus, most of the space just used by the heap for replacement selection (typically many blocks in length) is not being used by the merge process.

We can make better use of this space and at the same time greatly reduce the number of passes needed to merge the runs if we merge several runs at a time. Multiway merging is similar to two-way merging. If we have B runs to merge, with a block from each run available in memory, then the B-way merge algorithm simply looks at B values (the front-most value for each input run) and selects the smallest one to output. This value is removed from its run, and the process is repeated. When the current block for any run is exhausted, the next block from that run is read from disk. Figure 8.10 illustrates a multiway merge.

Conceptually, multiway merge assumes that each run is stored in a separate file. However, this is not necessary in practice. We only need to know the position of each run within a single file, and use **`seek`** to move to the appropriate block whenever we need new data from a particular run. Naturally, this approach destroys the ability to do sequential processing on the input file. However, if all runs were stored on a single disk drive, then processing would not be truly sequential anyway because the I/O head would be alternating between the runs. Thus, multiway merging replaces several (potentially) sequential passes with a single random access pass. If

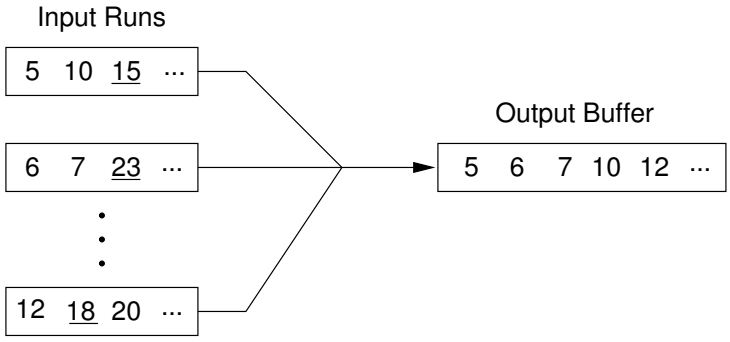

Figure 8.10 Illustration of multiway merge. The first value in each input run is examined and the smallest sent to the output. This value is removed from the input and the process repeated. In this example, values 5, 6, and 12 are compared first. Value 5 is removed from the first run and sent to the output. Values 10, 6, and 12 will be compared next. After the first five values have been output, the "current" value of each block is the one underlined.

the processing would not be sequential anyway (such as when all processing is on a single disk drive), no time is lost by doing so.

Multiway merging can greatly reduce the number of passes required. If there is room in memory to store one block for each run, then all runs can be merged in a single pass. Thus, replacement selection can build initial runs in one pass, and multiway merging can merge all runs in one pass, yielding a total cost of two passes. However, for truly large files, there might be too many runs for each to get a block in memory. If there is room to allocate B blocks for a B-way merge, and the number of runs R is greater than B, then it will be necessary to do multiple merge passes. In other words, the first B runs are merged, then the next B, and so on. These super-runs are then merged by subsequent passes, B super-runs at a time.

How big a file can be merged in one pass? Assuming B blocks were allocated to the heap for replacement selection (resulting in runs of average length $2B$ blocks), followed by a B-way merge, we can process on average a file of size $2B^2$ blocks in a single multiway merge. $2B^{k+1}$ blocks on average can be processed in k B-way merges. To gain some appreciation for how quickly this grows, assume that we have available 0.5MB of working memory, and that a block is 4KB, yielding 128 blocks in working memory. The average run size is 1MB (twice the working memory size). In one pass, 128 runs can be merged. Thus, a file of size 128MB can, on average, be processed in two passes (one to build the runs, one to do the merge) with only 0.5MB of working memory. As another example, assume blocks are 1KB long and working memory is 1MB $=$ 1024 blocks. Then 1024 runs of average length 2MB (which is about 2GB) can be combined in a single merge pass. A larger block size would reduce the size of the file that can be processed

File Size (Mb)	Sort 1	Sort 2 Memory size (in blocks)				Sort 3 Memory size (in blocks)		
		2	4	16	256	2	4	16
1	0.61	0.27	0.24	0.19	0.10	0.21	0.15	0.13
	4,864	2,048	1,792	1,280	256	2,048	1,024	512
4	2.56	1.30	1.19	0.96	0.61	1.15	0.68	0.66*
	21,504	10,240	9,216	7,168	3,072	10,240	5,120	2,048
16	11.28	6.12	5.63	4.78	3.36	5.42	3.19	3.10
	94,208	49,152	45,056	36,864	20,480	49,152	24,516	12,288
256	220.39	132.47	123.68	110.01	86.66	115.73	69.31	68.71
	1,769K	1,048K	983K	852K	589K	1,049K	524K	262K

Figure 8.11 A comparison of three external sorts on a collection of small records for files of various sizes. Each entry in the table shows time in seconds and total number of blocks read and written by the program. File sizes are in Megabytes. For the third sorting algorithm, on a file size of 4MB, the time and blocks shown in the last column are for a 32-way merge (marked with an asterisk). 32 is used instead of 16 because 32 is a root of the number of blocks in the file (while 16 is not), thus allowing the same number of runs to be merged at every pass.

in one merge pass for a fixed-size working memory; a smaller block size or larger working memory would increase the file size that can be processed in one merge pass. Two merge passes allow much bigger files to be processed. With 0.5MB of working memory and 4KB blocks, a file of size 16 gigabytes could be processed in two merge passes, which is big enough for most applications. Thus, this is a very effective algorithm for single disk drive external sorting.

Figure 8.11 shows a comparison of the running time to sort various-sized files for the following implementations: (1) standard Mergesort with two input runs and two output runs, (2) two-way Mergesort with large initial runs (limited by the size of available memory), and (3) R-way Mergesort performed after generating large initial runs. In each case, the file was composed of a series of four-byte records (a two-byte key and a two-byte data value), or 256K records per megabyte of file size. We can see from this table that using even a modest memory size (two blocks) to create initial runs results in a tremendous savings in time. Doing 4-way merges of the runs provides another considerable speedup, however large-scale multi-way merges for R beyond about 4 or 8 runs does not help much because a lot of time is spent determining which is the next smallest element among the R runs.

We see from this experiment that building large initial runs reduces the running time to slightly more than one third that of standard Mergesort, depending on file and memory sizes. Using a multiway merge further cuts the time nearly in half.

In summary, a good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing, and output as much as possible.

- Use as much working memory as possible. Applying more memory usually speeds processing. In fact, more memory will have a greater effect than a faster disk. A faster CPU is unlikely to yield much improvement in running time for external sorting, because disk I/O speed is the limiting factor.
- If possible, use additional disk drives for more overlapping of processing with I/O, and to allow for sequential file processing.

8.6 Further Reading

A good general text on file processing is Folk and Zoellick's *File Structures: A Conceptual Toolkit* [FZ98]. A somewhat more advanced discussion on key issues in file processing is Betty Salzberg's *File Structures: An Analytical Approach* [Sal88]. A great discussion on external sorting methods can be found in Salzberg's book. The presentation in this chapter is similar in spirit to Salzberg's.

For details on disk drive modeling and measurement, see the article by Ruemmler and Wilkes, "An Introduction to Disk Drive Modeling" [RW94]. See Andrew S. Tanenbaum's *Structured Computer Organization* [Tan06] for an introduction to computer hardware and organization. An excellent, detailed description of memory and hard disk drives can be found online at "The PC Guide," by Charles M. Kozierok [Koz05] (`www.pcguide.com`). The PC Guide also gives detailed descriptions of the Microsoft Windows and UNIX (Linux) file systems.

See "Outperforming LRU with an Adaptive Replacement Cache Algorithm" by Megiddo and Modha [MM04] for an example of a more sophisticated algorithm than LRU for managing buffer pools.

The snowplow argument comes from Donald E. Knuth's *Sorting and Searching* [Knu98], which also contains a wide variety of external sorting algorithms.

8.7 Exercises

8.1 Computer memory and storage prices change rapidly. Find out what the current prices are for the media listed in Figure 8.1. Does your information change any of the basic conclusions regarding disk processing?

8.2 Assume a disk drive from the late 1990s is configured as follows. The total storage is approximately 675MB divided among 15 surfaces. Each surface has 612 tracks; there are 144 sectors/track, 512 bytes/sector, and 8 sectors/cluster. The disk turns at 3600 rpm. The track-to-track seek time is 20 ms, and the average seek time is 80 ms. Now assume that there is a 360KB file on the disk. On average, how long does it take to read all of the data in the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on adjacent tracks, and that the file completely fills each track on which it is found. A seek must be performed each time the I/O head moves to a new track. Show your calculations.

8.3 Using the specifications for the disk drive given in Exercise 8.2, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.4 Using the disk drive specifications given in Exercise 8.2, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 4KB clusters.

Show your calculations.

8.5 Assume that a disk drive is configured as follows. The total storage is approximately 1033MB divided among 15 surfaces. Each surface has 2100 tracks, there are 64 sectors/track, 512 bytes/sector, and 8 sectors/cluster. The disk turns at 7200 rpm. The track-to-track seek time is 3 ms, and the average seek time is 20 ms. Now assume that there is a 512KB file on the disk. On average, how long does it take to read all of the data on the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

8.6 Using the specifications for the disk drive given in Exercise 8.5, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.7 Using the disk drive specifications given in Exercise 8.5, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 4KB clusters.

Show your calculations.

8.8 A typical disk drive from 2004 has the following specifications.[3] The total storage is approximately 120GB on 6 platter surfaces or 20GB/platter. Each platter has 16K tracks with 2560 sectors/track (a sector holds 512 bytes) and 16 sectors/cluster. The disk turns at 7200 rpm. The track-to-track seek time is 2.0 ms, and the average seek time is 10.0 ms. Now assume that there is a 6MB file on the disk. On average, how long does it take to read all of the data on the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

[3]To make the exercise doable, this specification is completely fictitious with respect to the track and sector layout. While sectors do have 512 bytes, and while the number of platters and amount of data per track is plausible, the reality is that all modern drives use a zoned organization to keep the data density from inside to outside of the disk reasonably high. The rest of the numbers are typical for a drive from 2004.

8.9 Using the specifications for the disk drive given in Exercise 8.8, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.10 Using the disk drive specifications given in Exercise 8.8, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 8KB clusters.

Show your calculations.

8.11 At the end of 2004, the fastest disk drive I could find specifications for was the Maxtor Atlas. This drive had a nominal capacity of 73.4GB using 4 platters (8 surfaces) or 9.175GB/surface. Assume there are 16,384 tracks with an average of 1170 sectors/track and 512 bytes/sector.[4] The disk turns at 15,000 rpm. The track-to-track seek time is 0.4 ms and the average seek time is 3.6 ms. How long will it take on average to read a 6MB file, assuming that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

8.12 Using the specifications for the disk drive given in Exercise 8.11, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.13 Using the disk drive specifications given in Exercise 8.11, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 8KB clusters.

Show your calculations.

8.14 Prove that two tracks selected at random from a disk are separated on average by one third the number of tracks on the disk.

8.15 Assume that a file contains one million records sorted by key value. A query to the file returns a single record containing the requested key value. Files are stored on disk in sectors each containing 100 records. Assume that the average time to read a sector selected at random is 10.0 ms. In contrast, it takes only 2.0 ms to read the sector adjacent to the current position of the I/O head. The "batch" algorithm for processing queries is to first sort the queries by order of appearance in the file, and then read the entire file sequentially, processing all queries in sequential order as the file is read. This algorithm implies that the queries must all be available before processing begins. The "interactive" algorithm is to process each query in order of its arrival, searching for the requested sector each time (unless by chance two queries in a row

[4] Again, this track layout does does not account for the zoned arrangement on modern disk drives.

are to the same sector). Carefully define under what conditions the batch method is more efficient than the interactive method.

8.16 Assume that a virtual memory is managed using a buffer pool. The buffer pool contains five buffers and each buffer stores one block of data. Memory accesses are by block ID. Assume the following series of memory accesses takes place:

5 2 5 12 3 6 5 9 3 2 4 1 5 9 8 15 3 7 2 5 9 10 4 6 8 5

For each of the following buffer pool replacement strategies, show the contents of the buffer pool at the end of the series, and indicate how many times a block was found in the buffer pool (instead of being read into memory). Assume that the buffer pool is initially empty.

(a) First-in, first out.
(b) Least frequently used (with counts kept only for blocks currently in memory, counts for a page are lost when that page is removed, and the oldest item with the smallest count is removed when there is a tie).
(c) Least frequently used (with counts kept for all blocks, and the oldest item with the smallest count is removed when there is a tie).
(d) Least recently used.
(e) Most recently used (replace the block that was most recently accessed).

8.17 Suppose that a record is 32 bytes, a block is 1024 bytes (thus, there are 32 records per block), and that working memory is 1MB (there is also additional space available for I/O buffers, program variables, etc.). What is the *expected* size for the largest file that can be merged using replacement selection followed by a *single* pass of multiway merge? Explain how you got your answer.

8.18 Assume that working memory size is 256KB broken into blocks of 8192 bytes (there is also additional space available for I/O buffers, program variables, etc.). What is the *expected* size for the largest file that can be merged using replacement selection followed by *two* passes of multiway merge? Explain how you got your answer.

8.19 Prove or disprove the following proposition: Given space in memory for a heap of M records, replacement selection will completely sort a file if no record in the file is preceded by M or more keys of greater value.

8.20 Imagine a database containing ten million records, with each record being 100 bytes long. Provide an estimate of the time it would take (in seconds) to sort the database on a typical desktop or laptop computer.

8.21 Assume that a company has a computer configuration satisfactory for processing their monthly payroll. Further assume that the bottleneck in payroll processing is a sorting operation on all of the employee records, and that

an external sorting algorithm is used. The company's payroll program is so good that it plans to hire out its services to do payroll processing for other companies. The president has an offer from a second company with 100 times as many employees. She realizes that her computer is not up to the job of sorting 100 times as many records in an acceptable amount of time. Describe what impact each of the following modifications to the computing system is likely to have in terms of reducing the time required to process the larger payroll database.

(a) A factor of two speedup to the CPU.
(b) A factor of two speedup to disk I/O time.
(c) A factor of two speedup to main memory access time.
(d) A factor of two increase to main memory size.

8.22 How can the external sorting algorithm described in this chapter be extended to handle variable-length records?

8.8 Projects

8.1 For a database application, assume it takes 10 ms to read a block from disk, 1 ms to search for a record in a block stored in memory, and that there is room in memory for a buffer pool of 5 blocks. Requests come in for records, with the request specifying which block contains the record. If a block is accessed, there is a 10% probability for each of the next ten requests that the request will be to the same block. What will be the expected performance improvement for each of the following modifications to the system?

(a) Get a CPU that is twice as fast.
(b) Get a disk drive that is twice as fast.
(c) Get enough memory to double the buffer pool size.

Write a simulation to analyze this problem.

8.2 Pictures are typically stored as an array, row by row, on disk. Consider the case where the picture has 16 colors. Thus, each pixel can be represented using 4 bits. If you allow 8 bits per pixel, no processing is required to unpack the pixels (because a pixel corresponds to a byte, the lowest level of addressing on most machines). If you pack two pixels per byte, space is saved but the pixels must be unpacked. Which takes more time to read from disk and access every pixel of the image: 8 bits per pixel, or 4 bits per pixel with 2 pixels per byte? Program both and compare the times.

8.3 Implement a disk-based buffer pool class based on the LRU buffer pool replacement strategy. Disk blocks are numbered consecutively from the beginning of the file with the first block numbered as 0. Assume that blocks are

4096 bytes in size, with the first 4 bytes used to store the block ID corresponding to that buffer. Use the first **`BufferPool`** abstract class given in Section 8.3 as the basis for your implementation.

8.4 Implement an external sort based on replacement selection and multiway merging as described in this chapter. Test your program both on files with small records and on files with large records. For what size record do you find that key sorting would be worthwhile?

8.5 Implement a Quicksort for large files on disk by replacing all array access in the normal Quicksort application with access to a virtual array implemented using a buffer pool. That is, whenever a record in the array would be read or written by Quicksort, use a call to a buffer pool function instead. Compare the running time of this implementation with implementations for external sorting based on mergesort as described in this chapter.

8.6 Section 8.5.1 suggests that an easy modification to the basic 2-way mergesort is to read in a large chunk of data into main memory, sort it with Quicksort, and write it out for initial runs. Then, a standard 2-way merge is used in a series of passes to merge the runs together. However, this makes use of only two blocks of working memory at a time. Each block read is essentially random access, because the various files are read in an unknown order, even though each of the input and output files is processed sequentially on each pass. A possible improvement would be, on the merge passes, to divide working memory into four equal sections. One section is allocated to each of the two input files and two output files. All reads during merge passes would be in full sections, rather than single blocks. While the total number of blocks read and written would be the same as a regular 2-way Mergesort, it is possible that this would speed processing because a series of blocks that are logically adjacent in the various input and output files would be read/written each time. Implement this variation, and compare its running time against a standard series of 2-way merge passes that read/write only a single block at a time. Before beginning implementation, write down your hypothesis on how the running time will be affected by this change. After implementing, did you find that this change has any meaningful effect on performance?