Chapter 6

Sorting

Concepts:
- ▷ Natural sorting techniques
- ▷ Recursive sorting techniques
- ▷ `Vector` sorting
- ▷ Use of `compareTo` methods
- ▷ `Comparator` techniques

"Come along, children. Follow me."
Before you could wink an eyelash
Jack, Knak, Lack, Mack,
Nack, Ouack, Pack, and Quack
fell into line, just as they had been taught.
—Robert McCloskey

COMPUTERS SPEND A CONSIDERABLE AMOUNT of their time keeping data in order. When we view a directory or folder, the items are sorted by name or type or modification date. When we search the Web, the results are returned sorted by "applicability." At the end of the month, our checks come back from the bank sorted by number, and our deposits are sorted by date. Clearly, in the grand scheme of things, sorting is an important function of computers. Not surprisingly, data structures can play a significant role in making sorts run quickly. This chapter begins an investigation of sorting methods.

6.1 Approaching the Problem

For the moment we assume that we will be sorting an unordered array of integers (see Figure 6.1a).[1] The problem is to arrange the integers so that every adjacent pair of values is in the correct order (see Figure 6.1b). A simple technique to sort the array is to pass through the array from left to right, swapping adjacent values that are out of order (see Figure 6.2). The exchange of values is accomplished with a utility method:

BubbleSort

```
public static void swap(int data[], int i, int j)
// pre: 0 <= i,j < data.length
// post: data[i] and data[j] are exchanged
{
    int temp;
```

[1] We focus on arrays of integers to maintain a simple approach. These techniques, of course, can be applied to vectors of objects, provided that some relative comparison can be made between two elements. This is discussed in Section 6.7.

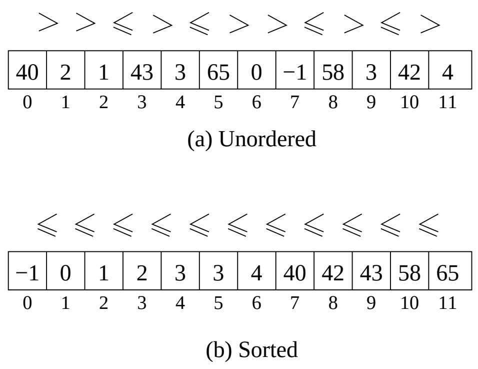


Figure 6.1 The relations between entries in unordered and sorted arrays of integers.

```
        temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
```

After a single pass the largest value will end up "bubbling" up to the high-indexed side of the array. The next pass will, at least, bubble up the next largest value, and so forth. The sort—called *bubble sort*—must be finished after $n - 1$ passes. Here is how we might write bubble sort in Java:

```
    public static void bubbleSort(int data[], int n)
    // pre: 0 <= n <= data.length
    // post: values in data[0..n-1] in ascending order
    {
        int numSorted = 0;      // number of values in order
        int index;              // general index
        while (numSorted < n)
        {
            // bubble a large element to higher array index
            for (index = 1; index < n-numSorted; index++)
            {
                if (data[index-1] > data[index])
                    swap(data,index-1,index);
            }
            // at least one more value in place
            numSorted++;
        }
    }
```

Observe that the only potentially time-consuming operations that occur in this sort are comparisons and exchanges. While the cost of comparing integers is rel-

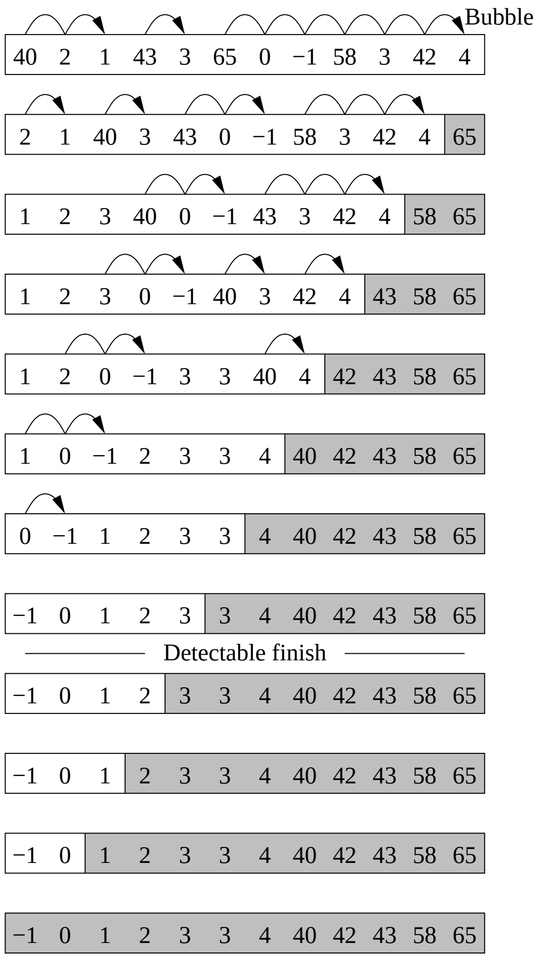


Figure 6.2 The passes of bubble sort: hops indicate "bubbling up" of large values. Shaded values are in sorted order. A pass with no exchanges indicates sorted data.

atively small, if each element of the array were to contain a long string (for example, a DNA sequence) or a complex object (for example, a Library of Congress entry), then the comparison of two values might be a computationally intensive operation. Similarly, the cost of performing an exchange is to be avoided.[2] We can, therefore, restrict our attention to the number of comparison and exchange operations that occur in sorts in order to adequately evaluate their performance.

In bubble sort each pass of the bubbling phase performs $n - 1$ comparisons and as many as $n-1$ exchanges. Thus the worst-case cost of performing bubble sort is $O((n-1)^2)$ or $O(n^2)$ operations. In the best case, none of the comparisons leads to an exchange. Even then, though, the algorithm has quadratic behavior.[3]

Most of us are inefficient sorters. Anyone having to sort a deck of cards or a stack of checks is familiar with the feeling that *there must be a better way to do this*. As we shall see, there probably is: most common sorting techniques used in day-to-day life run in $O(n^2)$ time, whereas the best single processor comparison-based sorting techniques are expected to run in only $O(n \log n)$ time. (If multiple processors are used, we can reduce this to $O(\log n)$ time, but that algorithm is beyond the scope of this text.) We shall investigate two sorting techniques that run in $O(n^2)$ time, on average, and two that run in $O(n \log n)$ time. In the end we will attempt to understand what makes the successful sorts successful.

Our first two sorting techniques are based on natural analogies.

6.2 Selection Sort

Children are perhaps the greatest advocates of *selection sort*. Every October, Halloween candies are consumed from best to worst. Whether daily sampling is limited or not, it is clear that choices of the next treat consumed are based on "the next biggest piece" or "the next-most favorite," and so on. Children consume treats in decreasing order of acceptability. Similarly, when we select plants from a greenhouse, check produce in the store, or pick strawberries from the farm we seek the best items first.

This selection process can be applied to an array of integers. Our goal is to identify the index of the largest element of the array. We begin by *assuming* that the first element is the largest, and then form a competition among all the remaining values. As we come across larger values, we update the index of the current maximum value. In the end, the index must point to the largest value. This code is idiomatic, so we isolate it here:

SelectionSort

```
int index;  // general index
int max;    // index of largest value
// determine maximum value in array
```

[2] In languages like Java, where large objects are manipulated through references, the cost of an exchange is usually fairly trivial. In many languages, however, the cost of exchanging large values stored directly in the array is a real concern.

[3] If, as we noted in Figure 6.2, we detected the lack of exchanges, bubble sort would run in $O(n)$ time on data that were already sorted. Still, the average case would be quadratic.

```
max = 0;
for (index = 1; index < numUnsorted; index++)
{
    if (data[max] < data[index]) max = index;
}
```

(Notice that the maximum is not updated unless a *larger* value is found.) Now, consider where this maximum value would be found if the data were sorted: it should be clear to the right, in the highest indexed location. This is easily accomplished: we simply swap the last element of the unordered array with the maximum. Once this swap is completed, we know that at least that one value is in the correct location, and we logically reduce the size of the problem by one. If we remove the $n-1$ largest values in successive passes (see Figure 6.3), we have selection sort. Here is how the entire method appears in Java:

```
public static void selectionSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0..n-1] are in ascending order
{
    int numUnsorted = n;
    int index;      // general index
    int max;        // index of largest value
    while (numUnsorted > 0)
    {
        // determine maximum value in array
        max = 0;
        for (index = 1; index < numUnsorted; index++)
        {
            if (data[max] < data[index]) max = index;
        }
        swap(data,max,numUnsorted-1);
        numUnsorted--;
    }
}
```

Selection sort potentially performs far fewer exchanges than bubble sort: selection sort performs exactly one per pass, while bubble sort performs as many as $n-1$. Like bubble sort, however, selection sort demands $O(n^2)$ time for comparisons.

Interestingly, the performance of selection sort is independent of the order of the data: if the data are already sorted, it takes selection sort just as long to sort as if the data were unsorted. We can improve on this behavior through a slightly different analogy.

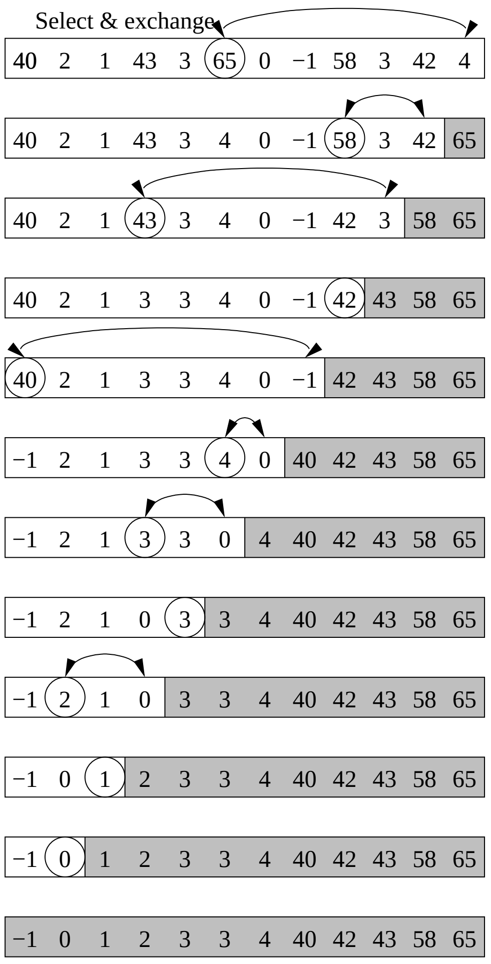


Figure 6.3 Profile of the passes of selection sort: shaded values are sorted. Circled values are maximum among unsorted values and are moved to the low end of sorted values on each pass.

6.3 Insertion Sort

Card players, when collecting a hand, often consider cards one at a time, inserting each into its sorted location. If we consider the "hand" to be the sorted portion of the array, and the "table" to be the unsorted portion, we develop a new sorting technique called *insertion sort*.

In the following Java implementation of insertion sort, the sorted values are kept in the low end of the array, and the unsorted values are found at the high end (see Figure 6.4). The algorithm consists of several "passes" of inserting the lowest-indexed unsorted value into the list of sorted values. Once this is done, of course, the list of sorted values increases by one. This process continues until each of the unsorted values has been incorporated into the sorted portion of the array. Here is the code:

InsertionSort

```
public static void insertionSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0..n-1] are in ascending order
{
    int numSorted = 1;      // number of values in place
    int index;              // general index
    while (numSorted < n)
    {
        // take the first unsorted value
        int temp = data[numSorted];
        // ...and insert it among the sorted:
        for (index = numSorted; index > 0; index--)
        {
            if (temp < data[index-1])
            {
                data[index] = data[index-1];
            } else {
                break;
            }
        }
        // reinsert value
        data[index] = temp;
        numSorted++;
    }
}
```

A total of $n - 1$ passes are made over the array, with a new unsorted value inserted each time. The value inserted may not be a new minimum or maximum value. Indeed, if the array was initially unordered, the value will, on average, end up near the middle of the previously sorted values. On random data the running time of insertion sort is expected to be dominated by $O(n^2)$ compares and data movements (most of the compares will lead to the movement of a data value).

If the array is initially in order, one compare is needed at every pass to determine that the value is already in the correct location. Thus, the inner loop

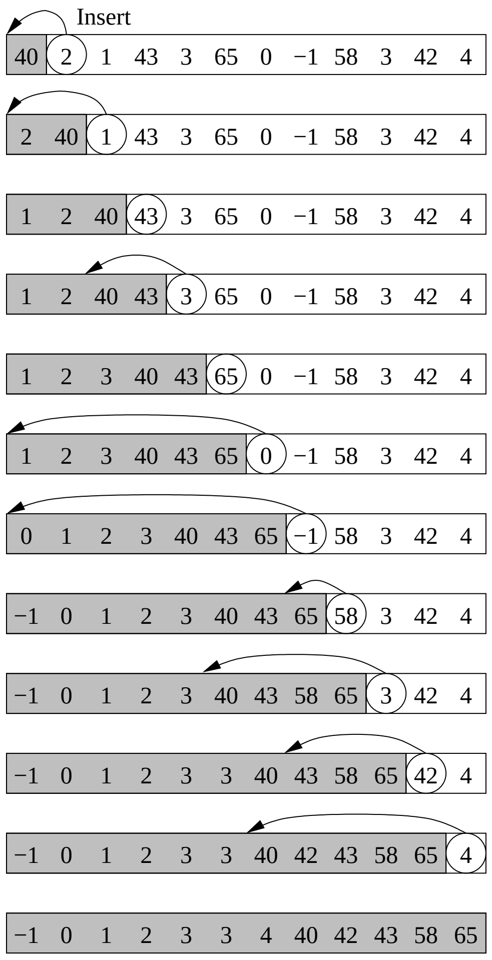


Figure 6.4 Profile of the passes of insertion sort: shaded values form a “hand” of sorted values. Circled values are successively inserted into the hand.

is executed exactly once for each of $n - 1$ passes. The best-case running time performance of the sort is therefore dominated by $O(n)$ comparisons (there are no movements of data within the array). Because of this characteristic, insertion sort is often used when data are very nearly ordered (imagine sorting a phone book after a month of new customers has been appended).

In contrast, if the array was previously in reverse order, the value must be compared with *every* sorted value to find the correct location. As the comparisons are made, the larger values are moved to the right to make room for the new value. The result is that each of $O(n^2)$ compares leads to a data movement, and the worst-case running time of the algorithm is $O(n^2)$.

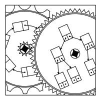
Note that each of these sorts uses a linear number of data cells. Not every sorting technique is able to live within this constraint.

6.4 Mergesort

Suppose that two friends are to sort an array of values. One approach might be to divide the deck in half. Each person then sorts one of two half-decks. The sorted deck is then easily constructed by combining the two sorted half-decks. This careful interleaving of sorted values is called a *merge*.

It is straightforward to see that a merge takes at least $O(n)$ time, because every value has to be moved into the destination deck. Still, within $n - 1$ comparisons, the merge must be finished. Since each of the $n - 1$ comparisons (and potential movements of data) takes at most constant time, the merge is no worse than linear.

There are, of course, some tricky aspects to the merge operation—for example, it is possible that all the cards in one half-deck are smaller than all the cards in the other. Still, the performance of the following merge code is $O(n)$:


MergeSort

```
private static void merge(int data[], int temp[],
                          int low, int middle, int high)
// pre: data[middle..high] are ascending
//      temp[low..middle-1] are ascending
// post: data[low..high] contains all values in ascending order
{
    int ri = low; // result index
    int ti = low; // temp index
    int di = middle; // destination index
    // while two lists are not empty merge smaller value
    while (ti < middle && di <= high)
    {
        if (data[di] < temp[ti]) {
            data[ri++] = data[di++]; // smaller is in high data
        } else {
            data[ri++] = temp[ti++]; // smaller is in temp
        }
    }
    // possibly some values left in temp array
```

```
        while (ti < middle)
        {
            data[ri++] = temp[ti++];
        }
        // ...or some values left (in correct place) in data array
    }
```

This code is fairly general, but a little tricky to understand (see Figure 6.5). We assume that the data from the two lists are located in the two arrays—in the lower half of the range in `temp` and in the upper half of the range in `data` (see Figure 6.5a). The first loop compares the first remaining element of each list to determine which should be copied over to the result list first (Figure 6.5b). That loop continues until one list is emptied (Figure 6.5c). If `data` is the emptied list, the remainder of the `temp` list is transferred (Figure 6.5d). If the `temp` list was emptied, the remainder of the `data` list is already located in the correct place!

Returning to our two friends, we note that before the two lists are merged each of the two friends is faced with sorting half the cards. How should this be done? If a deck contains fewer than two cards, it's already sorted. Otherwise, each person could recursively hand off half of his or her respective deck (now one-fourth of the entire deck) to a new individual. Once these small sorts are finished, the quarter decks are merged, finishing the sort of the half decks, and the two half decks are merged to construct a completely sorted deck. Thus, we might consider a new sort, called *mergesort*, that recursively splits, sorts, and reconstructs, through merging, a deck of cards. The logical "phases" of mergesort are depicted in Figure 6.6.

```
    private static void mergeSortRecursive(int data[],
                                           int temp[],
                                           int low, int high)
    // pre: 0 <= low <= high < data.length
    // post: values in data[low..high] are in ascending order
    {
        int n = high-low+1;
        int middle = low + n/2;
        int i;

        if (n < 2) return;
        // move lower half of data into temporary storage
        for (i = low; i < middle; i++)
        {
            temp[i] = data[i];
        }
        // sort lower half of array
        mergeSortRecursive(temp,data,low,middle-1);
        // sort upper half of array
        mergeSortRecursive(data,temp,middle,high);
        // merge halves together
        merge(data,temp,low,middle,high);
    }
```

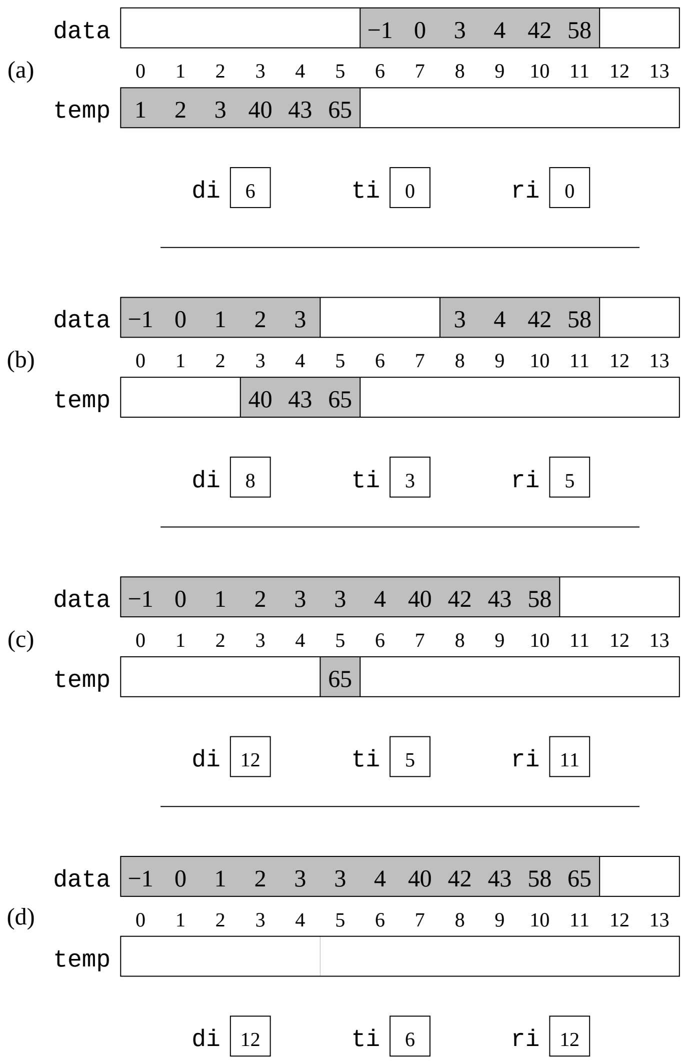


Figure 6.5 Four stages of a merge of two six element lists (shaded entries are participating values): (a) the initial location of data; (b) the merge of several values; (c) the point at which a list is emptied; and (d) the final result.

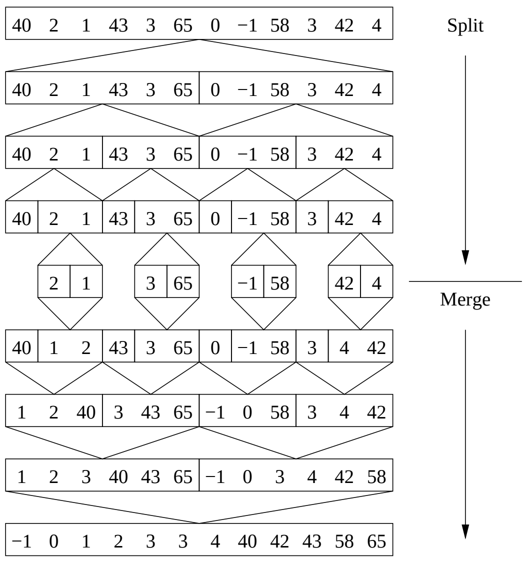


Figure 6.6 Profile of mergesort: values are recursively split into unsorted lists that are then recursively merged into ascending order.

Note that this sort requires a temporary array to perform the merging. This temporary array is only used by a single merge at a time, so it is allocated once and garbage collected after the sort. We hide this detail with a public wrapper procedure that allocates the array and calls the recursive sort:

```
public static void mergeSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0..n-1] are in ascending order
{
    mergeSortRecursive(data,new int[n],0,n-1);
}
```

Clearly, the depth of the splitting is determined by the number of times that n can be divided in two and still have a value of 1 or greater: $\log_2 n$. At each level of splitting, every value must be merged into its respective subarray. It follows that at each logical level, there are $O(n)$ compares over all the merges. Since there are $\log_2 n$ levels, we have $O(n \cdot \log n)$ units of work in performing a mergesort.

Mergesort is a common technique for sorting large sets of data that do not fit completely in fast memory. Instead, the data are saved in temporary files that are merged together. When the recursion splits the collection into subsets of a manageable size, they can be sorted using other techniques, if desired.

One of the unappealing aspects of mergesort is that it is difficult to merge two lists without significant extra memory. If we could avoid the use of this extra space without significant increases in the number of comparisons or data movements, then we would have an excellent sorting technique. Our next method demonstrates an $O(n \log n)$ method that requires significantly less space.

6.5 Quicksort

Since the process of sorting numbers consists of moving each value to its ultimate location in the sorted array, we might make some progress toward a solution if we could move *a single value* to its ultimate location. This idea forms the basis of a fast sorting technique called *quicksort*.

One way to find the correct location of, say, the leftmost value—called a *pivot*—in an unsorted array is to rearrange the values so that all the smaller values appear to the left of the pivot, and all the larger values appear to the right. One method of partitioning the data is shown here. It returns the final location for what was originally the leftmost value:

QuickSort

```
private static int partition(int data[], int left, int right)
// pre: left <= right
// post: data[left] placed in the correct (returned) location
{
    while (true)
    {
        // move right "pointer" toward left
        while (left < right && data[left] < data[right]) right--;
        if (left < right) swap(data,left++,right);
        else return left;
        // move left pointer toward right
        while (left < right && data[left] < data[right]) left++;
        if (left < right) swap(data,left,right--);
        else return right;
    }
}
```

The indices `left` and `right` start at the two ends of the array (see Figure 6.7) and move toward each other until they coincide. The pivot value, being leftmost in the array, is indexed by `left`. Everything to the left of `left` is smaller than the pivot, while everything to the right of `right` is larger. Each step of the main loop compares the left and right values and, if they're out of order, exchanges them. Every time an exchange occurs the index (`left` or `right`) that references the pivot value is alternated. In any case, the nonpivot variable is moved toward the other. Since, at each step, `left` and `right` move one step closer to each other, within n steps, `left` and `right` are equal, and they point to the current location of the pivot value. Since only smaller values are to the left of the pivot, and larger values are to the right, the pivot must be located in its final location. Values correctly located are shaded in Figure 6.8.

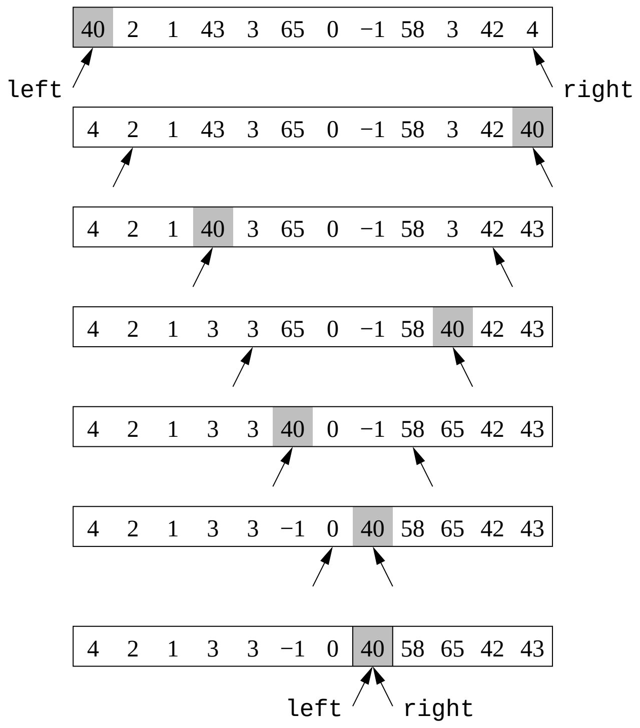


Figure 6.7 The partitioning of an array's values based on the (shaded) pivot value 40. Snapshots depict the state of the data after the `if` statements of the `partition` method.

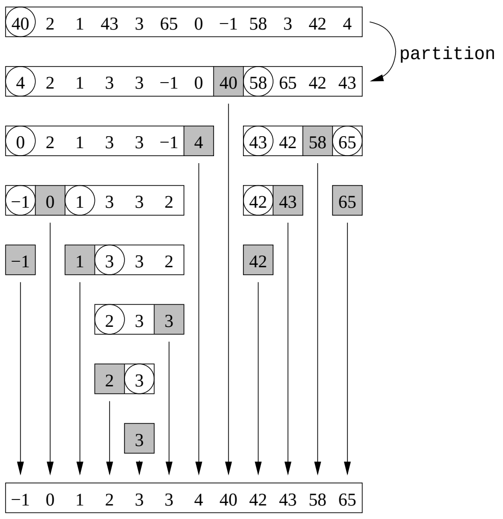


Figure 6.8 Profile of quicksort: leftmost value (the circled *pivot*) is used to position value in final location (indicated by shaded) and partition array into relatively smaller and larger values. Recursive application of partitioning leads to quicksort.

Because the pivot segregates the larger and smaller values, we know that none of these values will appear on the opposite side of the pivot in the final arrangement. This suggests that we can reduce the sorting of a problem of size n to two problems of size approximately $\frac{n}{2}$. To finish the sort, we need only recursively sort the values to the left and right of the pivot:

```
public static void quickSort(int data[], int n)
// post: the values in data[0..n-1] are in ascending order
{
    quickSortRecursive(data,0,n-1);
}

private static void quickSortRecursive(int data[],int left,int right)
// pre: left <= right
// post: data[left..right] in ascending order
{
    int pivot;   // the final location of the leftmost value
    if (left >= right) return;
    pivot = partition(data,left,right);    /* 1 - place pivot */
    quickSortRecursive(data,left,pivot-1); /* 2 - sort small */
    quickSortRecursive(data,pivot+1,right);/* 3 - sort large */
    /* done! */
}
```

In practice, of course, the splitting of the values is not always optimal (see the placement of the value 4 in Figure 6.8), but a careful analysis suggests that even with these “tough breaks” quicksort takes only $O(n \log n)$ time.

When either sorted or reverse-sorted data are to be sorted by quicksort, the results are disappointing. This is because the pivot value selected (here, the leftmost value) finds its ultimate location at one end of the array or the other. This reduces the sort of n values to $n - 1$ values (and *not* $n/2$), and the sort requires $O(n)$ passes of an $O(n)$ step partition. The result is an $O(n^2)$ sort. Since nearly sorted data are fairly common, this result is to be avoided.

Notice that picking the leftmost value is not special. If, instead, we attempt to find the correct location for the middle value, then other arrangements of data will cause the degenerate behavior. In short, for any specific or *deterministic* partitioning technique, a degenerate arrangement exists. The key to more consistent performance, then, is a *nondeterministic* partitioning that correctly places a value selected at random (see Problem 6.15). There is, of course, a very unlikely chance that the data are in order *and* the positions selected induce a degenerate behavior, but that chance is small and successive runs of the sorting algorithm on the same data are exceedingly unlikely to exhibit the same behavior. So, although the worst-case behavior is still $O(n^2)$, its expected behavior is $O(n \log n)$.

Quicksort is an excellent sort when data are to be sorted with little extra space. Because the speed of partitioning depends on the random access nature of arrays or `Vectors`, quicksort is not suitable when not used with random access structures. In these cases, however, other fast sorts are often possible.

6.6 Radix Sort

After investigating a number of algorithms that sort in $O(n^2)$ or $O(n \log n)$ time, one might wonder if it is possible to sort in linear time. If the right conditions hold, we can sort certain types of data in linear time. First, we must investigate a pseudogame, 52 pickup!

Suppose we drop a deck of 52 cards on the floor, and we want to not only pick them up, but we wish to sort them at the same time. It might be most natural to use an insertion sort: we keep a pile of sorted cards and, as we pick up new cards, we insert them in the deck in the appropriate position. A more efficient approach makes use of the fact that we know what the sorted deck looks like. We simply lay out the cards in a row, with each position in the row reserved for the particular card. As we pick up a card, we place it in its reserved location. In the end, all the cards are in their correct location and we collect them from left to right.

Exercise 6.1 *Explain why this sorting technique always takes $O(n)$ time for a deck of n cards.*

Such an approach is the basis for a general sorting technique called *bucket sort*. By quickly inspecting values (perhaps a word) we can approximately sort them into different buckets (perhaps based on the first letter). In a subsequent pass we can sort the values in each bucket with, perhaps a different sort. The buckets of sorted values are then accumulated, carefully maintaining the order of the buckets, and the result is completely sorted. Unfortunately, the worst-case behavior of this sorting technique is determined by the performance of the algorithm we use to sort each bucket of values.

Exercise 6.2 *Suppose we have n values and m buckets and we use insertion sort to perform the sort of each bucket. What is the worst-case time complexity of this sort?*

Such a technique can be used to sort integers, especially if we can partially sort the values based on a single digit. For example, we might develop a support function, `digit`, that, given a number `n` and a decimal place `d`, returns the value of the digit in the particular decimal place. If `d` was 0, it would return the units digit of `n`. Here is a recursive implementation:

RadixSort

```
public static int digit(int n, int d)
// pre: n >= 0 and d >= 0
// post: returns the value of the dth decimal place of n
//  where the units place has position 0
{
    if (d == 0) return n % 10;
    else return digit(n/10,d-1);
}
```

Here is the code for placing an array of integer values among 10 buckets, based on the value of digit `d`. For example, if we have numbers between 1 and 52 and we set `d` to 2, this code almost sorts the values based on their 10's digit.

```
public static void bucketPass(int data[], int d)
// pre: data is an array of data values, and d >= 0
// post: data is sorted by the digit found in location d;
// if two values have the same digit in location d, their
// relative positions do not change; i.e., they are not swapped
{
    int i,j;
    int value;
    // allocate some buckets
    Vector<Vector<Integer>> bucket = new Vector<Vector<Integer>>(10);
    bucket.setSize(10);
    // allocate Vectors to hold values in each bucket
    for (j = 0; j < 10; j++) bucket.set(j,new Vector<Integer>());
    // distribute the data among buckets
    int n = data.length;
    for (i = 0; i < n; i++)
    {
        value = data[i];
        // determine the d'th digit of value
```

```
            j = digit(value,d);
            // add data value to end of vector; keeps values in order
            bucket.get(j).add(value);
        }
        // collect data from buckets back into array
        // collect in reverse order to unload Vectors
        // in linear time
        i = n;
        for (j = 9; j >= 0; j--)
        {
            // unload all values in bucket j
            while (!bucket.get(j).isEmpty())
            {
                i--;
                value = bucket.get(j).remove();
                data[i] = value;
            }
        }
    }
```

We now have the tools to support a new sort, *radix sort*. The approach is to use the bucketPass code to sort all the values based on the units place. Next, all the values are sorted based on their 10's digit. The process continues until enough passes have been made to consider all the digits. If it is known that values are bounded above, then we can also bound the number of passes as well. Here is the code to perform a radix sort of values under 1 million (six passes):

```
    public static void radixSort(int data[])
    // pre: data is array of values; each is less than 1,000,000
    // post: data in the array are sorted into increasing order
    {
        for (int i = 0; i < 6; i++)
        {
            bucketPass(data,i);
        }
    }
```

After the first bucketPass, the values are ordered, based on their units digit. All values that end in 0 are placed near the front of data (see Figure 6.9), all the values that end in 9 appear near the end. Among those values that end in 0, the values appear *in the order they originally appeared in the array.* In this regard, we can say that bucketPass is a *stable* sorting technique. All other things being equal, the values appear in their original order.

During the second pass, the values are sorted, based on their 10's digit. Again, if two values have the same 10's digit, the relative order of the values is maintained. That means, for example, that 140 will appear before 42, because after the first pass, the 140 appeared before the 42. The process continues, until

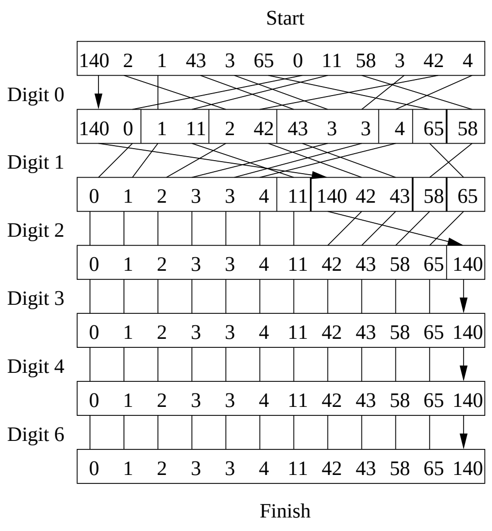


Figure 6.9 The state of the `data` array between the six passes of `radixSort`. The boundaries of the buckets are identified by vertical lines; bold lines indicate empty buckets. Since, on every pass, paths of incoming values to a bucket do not cross, the sort is stable. Notice that after three passes, the `radixSort` is finished. The same would be true, no matter the number of values, as long as they were all under 1000.

all digits are considered. Here, six passes are performed, but only three are necessary (see Problem 6.9).

There are several important things to remember about the construction of this sort. First, `bucketPass` is stable. That condition is necessary if the work of previous passes is not to be undone. Secondly, the sort is unlikely to work if the passes are performed from the most significant digit toward the units digit. Finally, since the number of passes is independent of the size of the `data` array, the speed of the entire sort is proportional to the speed of a single pass. Careful design allows the `bucketPass` to be accomplished in $O(n)$ time. We see, then, that `radixSort` is a $O(n)$ sorting method.

While, theoretically, `radixSort` can be accomplished in linear time, practically, the constant of proportionality associated with the bound is large compared to the other sorts we have seen in this chapter. In practice, `radixSort` is inefficient compared to most other sorts.

6.7 Sorting Objects

Sorting arrays of integers is suitable for understanding the performance of various sorts, but it is hardly a real-world problem. Frequently, the object that needs to be sorted is an `Object` with many fields, only some of which are actually used in making a comparison.

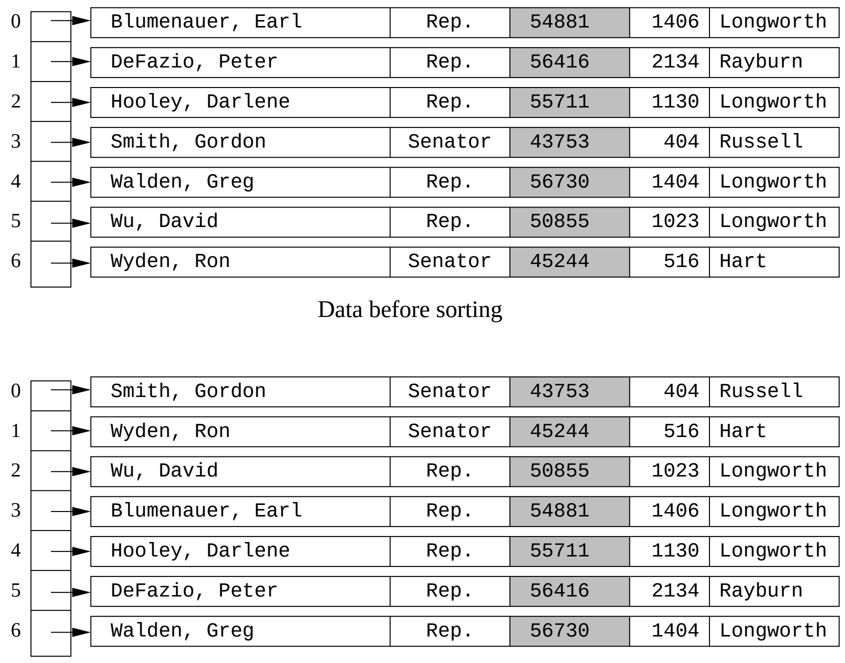
Let's consider the problem of sorting the entries associated with an electronic phone book. The first step is to identify the structure of a single entry in the phone book. Perhaps it has the following form:

PhoneBook

```
class PhoneEntry
{
    String name;        // person's name
    String title;       // person's title
    int extension;      // telephone number
    int room;           // number of room
    String building;    // office building

    public PhoneEntry(String n, String t, int e,
                      String b, int r)
    // post: construct a new phone entry
    {
      ...
    }

    public int compareTo(PhoneEntry other)
    // pre: other is non-null
    // post: returns integer representing relation between values
    {
        return this.extension - other.extension;
    }
}
```



0	Blumenauer, Earl	Rep.	54881	1406	Longworth
1	DeFazio, Peter	Rep.	56416	2134	Rayburn
2	Hooley, Darlene	Rep.	55711	1130	Longworth
3	Smith, Gordon	Senator	43753	404	Russell
4	Walden, Greg	Rep.	56730	1404	Longworth
5	Wu, David	Rep.	50855	1023	Longworth
6	Wyden, Ron	Senator	45244	516	Hart

Data before sorting

0	Smith, Gordon	Senator	43753	404	Russell
1	Wyden, Ron	Senator	45244	516	Hart
2	Wu, David	Rep.	50855	1023	Longworth
3	Blumenauer, Earl	Rep.	54881	1406	Longworth
4	Hooley, Darlene	Rep.	55711	1130	Longworth
5	DeFazio, Peter	Rep.	56416	2134	Rayburn
6	Walden, Greg	Rep.	56730	1404	Longworth

Data after sorting by telephone

Figure 6.10 An array of phone entries for the 107th Congressional Delegation from Oregon State, before and after sorting by telephone (shaded).

We have added the `compareTo` method to describe the relation between two entries in the phone book (the shaded fields of Figure 6.10). The `compareTo` method returns an integer that is less than, equal to, or greater than 0 when `this` is logically less than, equal to, or greater than `other`. We can now modify any of the sort techniques provided in the previous section to sort an array of phone entries:

```
public static void insertionSort(PhoneEntry data[], int n)
// pre: n <= data.length
// post: values in data[0..n-1] are in ascending order
{
    int numSorted = 1;       // number of values in place
    int index;               // general index
    while (numSorted < n)
    {
        // take the first unsorted value
        PhoneEntry temp = data[numSorted];
        // ...and insert it among the sorted:
            for (index = numSorted; index > 0; index--)
            {
                if (temp.compareTo(data[index-1]) < 0)
                {
                    data[index] = data[index-1];
                } else {
                    break;
                }
            }
            // reinsert value
            data[index] = temp;
            numSorted++;
        }
    }
```

Careful review of this insertion sort routine shows that all the $<$ operators have been replaced by checks for negative `compareTo` values. The result is that the phone entries in the array are ordered by increasing phone number.

If two or more people use the same extension, then the order of the resulting entries depends on the stability of the sort. If the sort is stable, then the relative order of the phone entries with identical extensions in the sorted array is the same as their relative order in the unordered array. If the sort is not stable, no guarantee can be made. To ensure that entries are, say, sorted in increasing order by extension and, in case of shared phones, sorted by increasing name, the following `compareTo` method might be used:

```
public int compareTo(PhoneEntry other)
// pre: other is non-null
// post: returns integer representing relation between values
{
    if (this.extension != other.extension)
        return this.extension - other.extension;
    else return this.name.compareTo(other.name);
}
```

Correctly specifying the relation between two objects with the `compareTo` method can be difficult when the objects cannot be *totally ordered*. Is it always possible that one athletic team is strictly less than another? Is it always the case that one set contains another? No. These are examples of domains that are *partially ordered*. Usually, however, most types may be totally ordered, and imagining how one might *sort* a collection of objects forces a suitable relation between any pair.

6.8 Ordering Objects Using Comparators

The definition of the `compareTo` method for an object should define, in a sense, the natural ordering of the objects. So, for example, in the case of a phone book, the entries would ideally be ordered based on the name associated with the entry. Sometimes, however, the `compareTo` method does not provide the ordering desired, or worse, the `compareTo` method has not been defined for an object. In these cases, the programmer turns to a simple method for specifing an external comparison method called a *comparator*. A comparator is an object that contains a method that is capable of comparing two objects. Sorting methods, then, can be developed to apply a comparator to two objects when a comparison is to be performed. The beauty of this mechanism is that different comparators can be applied to the same data to sort in different orders or on different keys. In Java a comparator is any class that implements the `java.util.Comparator` interface. This interface provides the following method:

Comparator

```
package java.util;
public interface Comparator
{
    public abstract int compare(Object a, Object b);
    // pre: a and b are valid objects, likely of similar type
    // post: returns a value less than, equal to, or greater than 0
    //       if a is less than, equal to, or greater than b
}
```

Like the `compareTo` method we have seen earlier, the `compare` method returns an integer that identifies the relationship between two values. Unlike the `compareTo` method, however, the `compare` method is not associated with the compared objects. As a result, the comparator is not privy to the implementation of the objects; it must perform the comparison based on information that is gained from accessor methods.

As an example of the implementation of a `Comparator`, we consider the implementation of a case-insensitive comparison of `String`s, called `Caseless-Comparator`. This comparison method converts both `String` objects to uppercase and then performs the standard `String` comparison:

Caseless-Comparator

```
public class CaselessComparator implements java.util.Comparator<String>
{
    public int compare(String a, String b)
    // pre: a and b are valid Strings
    // post: returns a value less than, equal to, or greater than 0
    //       if a is less than, equal to, or greater than b, without
    //       consideration of case
    {
        String upperA = ((String)a).toUpperCase();
        String upperB = ((String)b).toUpperCase();
        return upperA.compareTo(upperB);
    }
}
```

The result of the comparison is that strings that are spelled similarly in different cases appear together. For example, if an array contains the words of the children's tongue twister:

```
Fuzzy Wuzzy was a bear.
Fuzzy Wuzzy had no hair.
Fuzzy Wuzzy wasn’t fuzzy, wuzzy?
```

we would expect the words to be sorted into the following order:

```
a bear. Fuzzy Fuzzy Fuzzy fuzzy, had hair.
no was wasn’t Wuzzy Wuzzy Wuzzy wuzzy?
```

This should be compared with the standard ordering of `String` values, which would generate the following output:

```
Fuzzy Fuzzy Fuzzy Wuzzy Wuzzy Wuzzy a bear.
fuzzy, had hair. no was wasn’t wuzzy?
```

To use a `Comparator` in a sorting technique, we need only replace the use of `compareTo` methods with `compare` methods from a `Comparator`. Here, for example, is an insertion sort that makes use of a `Comparator` to order the values in an array of `Objects`:

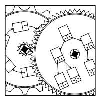
CompInsSort

```
public static <T> void insertionSort(T data[], Comparator<T> c)
// pre: c compares objects found in data
// post: values in data[0..n-1] are in ascending order
{
    int numSorted = 1;      // number of values in place
    int index;              // general index
    int n = data.length;    // length of the array
    while (numSorted < n)
    {
        // take the first unsorted value
        T temp = data[numSorted];
        // ...and insert it among the sorted:
        for (index = numSorted; index > 0; index--)
        {
            if (c.compare(temp,data[index-1]) < 0)
            {
                data[index] = data[index-1];
            } else {
                break;
            }
        }
        // reinsert value
        data[index] = temp;
        numSorted++;
    }
}
```

Note that in this description we don’t see the particulars of the types involved. Instead, all data are manipulated as `Objects`, which are specifically manipulated by the `compare` method of the provided `Comparator`.

6.9 Vector-Based Sorting

We extend the phone book example one more time, by allowing the PhoneEntrys to be stored in a Vector. There are, of course, good reasons to use Vector over arrays, but there are some added complexities that should be considered. Here is an alternative Java implementation of insertionSort that is dedicated to the sorting of a Vector of PhoneEntrys:


PhoneBook

```
protected void swap(int i, int j)
// pre: 0 <= i,j < this.size
// post: elements i and j are exchanged within the vector
{
    PhoneEntry temp;
    temp = get(i);
    set(i,get(j));
    set(j,temp);
}

public void insertionSort()
// post: values of vector are in ascending order
{
    int numSorted = 0;      // number of values in place
    int index;              // general index
    while (numSorted < size())
    {
        // take the first unsorted value
        PhoneEntry temp = (PhoneEntry)get(numSorted);
        // ...and insert it among the sorted:
        for (index = numSorted; index > 0; index--)
        {
            if (temp.compareTo((PhoneEntry)get(index-1)) < 0)
            {
                set(index,get(index-1));
            } else {
                break;
            }
        }
        // reinsert value
        set(index,temp);
        numSorted++;
    }
}
```

Recall that, for Vectors, we use the get method to fetch a value and set to store. Since any type of object may be referenced by a vector entry, we verify the type expected when a value is retrieved from the vector. This is accomplished through a parenthesized *cast*. If the type of the fetched value doesn't match the type of the cast, the program throws a *class cast exception*. Here, we cast the result of get in the compareTo method to indicate that we are comparing PhoneEntrys.

It is unfortunate that the `insertionSort` has to be specially coded for use with the `PhoneEntry` objects.

Exercise 6.3 *Write an* `insertionSort` *that uses a* `Comparator` *to sort a* `Vector` *of objects.*

6.10 Conclusions

Sorting is an important and common process on computers. In this chapter we considered several sorting techniques with quadratic running times. Bubble sort approaches the problem by checking and rechecking the relationships between elements. Selection and insertion sorts are based on techniques that people commonly use. Of these, insertion sort is most frequently used; it is easily coded and provides excellent performance when data are nearly sorted.

Two recursive sorting techniques, mergesort and quicksort, use recursion to achieve $O(n \log n)$ running times, which are optimal for comparison-based techniques on single processors. Mergesort works well in a variety of situations, but often requires significant extra memory. Quicksort requires a random access structure, but runs with little space overhead. Quicksort is not a stable sort because it has the potential to swap two values with equivalent keys.

We have seen with radix sort, it is possible to have a linear sorting algorithm, but it cannot be based on compares. Instead, the technique involves carefully ordering values based on looking at portions of the key. The technique is, practically, not useful for general-purpose sorting, although for many years, punched cards were efficiently sorted using precisely the method described here.

Sorting is, arguably, the most frequently executed algorithm on computers today. When we consider the notion of an *ordered structure*, we will find that algorithms and structures work hand in hand to help keep data in the correct order.

Self Check Problems

Solutions to these problems begin on page 445.

6.1 Why does it facilitate the `swap` method to have a temporary reference?

6.2 Cover the numbers below with your hand. Now, moving your hand to the right, expose each number in turn. On a separate sheet of paper, keep the list of values you have encounted in order. At the end you have sorted all of the values. Which sorting technique are you using?

296	457	-95	39	21	12	3.1	64	998	989

6.3 Copy the above table onto a piece of scrap paper. Start a column of numbers: write down the smallest table value you see into your column, crossing it out of the table. Continue until you have considered each of the values. What sorting technique are you using?

6.4 During spring cleaning, you decide to sort four months of checks returned with your bank statements. You decide to sort each month separately and go from there. Is this valid? If not, why. If it is, what happens next?

6.5 A postal employee approximately sorts mail into, say, 10 piles based on the magnitude of the street number of each address, pile 1 has 1-10, pile 2 has 11-20, etc. The letters are then collected together by increasing pile number. She then sorts them into a delivery crate with dividers labeled with street names. The order of streets corresponds to the order they appear on her mail route. What type of sort is she performing?

6.6 What is the purpose of the `compareTo` method?

Problems

Solutions to the odd-numbered problems begin on page 464.

6.1 Show that to exchange two integer values it is not strictly necessary to use a third, temporary integer variable. (Hint: Use addition and/or subtraction.)

6.2 We demonstrated that, in the worst case, bubble sort performs $O(n^2)$ operations. We assumed, however, that each pass performed approximately $O(n)$ operations. In fact, pass i performs as many as $O(n-i)$ operations, for $1 \leq i \leq n-1$. Show that bubble sort still takes $O(n^2)$ time.

6.3 How does `bubbleSort` (as presented) perform in the best and average cases?

6.4 On any pass of bubble sort, if no exchanges are made, then the relations between all the values are those desired, and the sort is done. Using this information, how fast will bubble sort run in worst, best, and average cases?

6.5 How fast does selection sort run in the best, worst, and average cases?

6.6 How fast does insertion sort run in the best, worst, and average cases? Give examples of best- and worst-case input for insertion sort.

6.7 Running an actual program, count the number of compares needed to sort n values using insertion sort, where n varies (e.g., powers of 2). Plot your data. Do the same thing for quicksort. Do the curves appear as theoretically expected? Does insertion sort ever run faster than quicksort? If so, at what point does it run slower?

6.8 Comparing insertion sort to quicksort, it appears that quicksort sorts more quickly without any increase in space. Is that true?

6.9 At the end of the discussion on radix sort, we pointed out that the digit sorting passes must occur from right to left. Give an example of an array of 5 two-digit values that do not sort properly if you perform the passes left to right.

6.10 In radix sort, it might be useful to terminate the sorting process when numbers do not change position during a call to `bucketPass`. Should this modification be adopted or not?

6.11 Using the millisecond timer, determine the length of time it takes to perform an assignment of a nonzero value to an `int`. (Hint: It will take less than a millisecond, so you will have to design several experiments that measure thousands or millions of assignments; see the previous lab, on page 115, for details.)

6.12 Running an actual program, and using the millisecond timer, `System.-currentTimeMillis`, measure the length of time needed to sort arrays of data of various sizes using a sort of your choice. Repeat the experiment but use `Vectors`. Is there a difference? In either case, explain why. (Hint: You may have to develop code along the lines of Problem 6.11.)

6.13 A sort is said to be *stable* if the order of equal values is maintained throughout the sort. Bubble sort is stable, because whenever two equal values are compared, no exchange occurs. Which other sorts are stable (consider insertion sort, selection sort, mergesort, and quicksort)?

6.14 The `partition` function of quicksort could be changed as follows: To place the leftmost value in the correct location, count the number of values that are strictly less than the leftmost value. The resulting number is the correct index for the desired value. Exchange the leftmost value for the value at the indexed location. With all other code left as it is, does this support a correctly functioning quicksort? If not, explain why.

6.15 Modify the `partition` method used by quicksort so that the pivot is randomly selected. (Hint: Before partitioning, consider placing the randomly selected value at the left side of the array.)

6.16 Write a recursive `selectionSort` algorithm. (Hint: Each level of recursion positions a value in the correct location.)

6.17 Write a recursive `insertionSort` algorithm.

6.18 Some of the best-performing sorts depend on the best-performing shuffles. A good shuffling technique rearranges data into any arrangement with equal probability. Design the most efficient shuffling mechanism you can, and argue its quality. What is its performance?

6.19 Write a program called `shuffleSort`. It first checks to see if the data are in order. If they are, the sort is finished. If they aren't, the data are shuffled and the process repeats. What is the best-case running time? Is there a worst-case running time? Why or why not? If each time the data were shuffled they were arranged in a never-seen-before configuration, would this change your answer?

6.20 Write a program to sort a list of unique integers between 0 and 1 million, but only using 1000 32-bit integers of space. The integers are read from a file.

6.11 Laboratory: Sorting with Comparators

Objective. To gain experience with Java's `java.util.Comparator` interface.

Discussion. In Chapter 6 we have seen a number of different sorting techniques. Each of the techniques demonstrated was based on the fixed, natural ordering of values found in an array. In this lab we will modify the `Vector` class so that it provides a method, `sort`, that can be used—with the help of a `Comparator`—to order the elements of the `Vector` in any of a number of different ways.

Procedure. Develop an extension of `structure.Vector`, called `MyVector`, that includes a new method, `sort`.

Here are some steps toward implementing this new class:

1. Create a new class, `MyVector`, which is declared to be an extension of the `structure.Vector` class. You should write a default constructor for this class that simply calls `super();`. This will force the `structure.Vector` constructor to be called. This, in turn, will initialize the protected fields of the `Vector` class.

2. Construct a new `Vector` method called `sort`. It should have the following declaration:

   ```
   public void sort(Comparator<T> c)
   // pre: c is a valid comparator
   // post: sorts this vector in order determined by c
   ```

   This method uses a `Comparator` type object to actually perform a sort of the values in `MyVector`. You may use any sort that you like.

3. Write an application that reads in a data file with several fields, and, depending on the `Comparator` used, sorts and prints the data in different orders.

Thought Questions. Consider the following questions as you complete the lab:

1. Suppose we write the following `Comparator`:

   ```
   import structure5.*;
   import java.util.Iterator;
   import java.util.Comparator;
   import java.util.Scanner;
   public class RevComparator<T> implements Comparator<T>
   {
       protected Comparator<T> base;

       public RevComparator(Comparator<T> baseCompare)
       {
           base = baseCompare;
   ```

```
    }

    public int compare(T a, T b)
    {
        return -base.compare(a,b);
    }
}
```

What happens when we construct:

```
MyVector<Integer> v = new MyVector<Integer>();
Scanner s = new Scanner(System.in);

while (s.hasNextInt())
{
    v.add(s.nextInt());
}

Comparator<Integer> c = new RevComparator<Integer>(new IntegerComparator());
v.sort(c);
```

2. In our examples, here, a new `Comparator` is necessary for each sorting order. How might it be possible to add state information (`protected` data) to the `Comparator` to allow it to sort in a variety of different ways? One might imagine, for example, a method called `ascending` that sets the `Comparator` to sort into increasing order. The `descending` method would set the `Comparator` to sort in reverse order.

Notes: