CHAPTER 12

SORTING AND SEARCHING

CHAPTER GOALS

To compare the selection sort and merge sort algorithms

To study the linear search and binary search algorithms

To appreciate that algorithms for the same task can differ widely in performance

To understand the big-Oh notation

To be able to estimate and compare the performance of algorithms

To write code to measure the running time of a program

CHAPTER CONTENTS

One of the most common tasks in data processing is sorting. For example, an array of employees often needs to be displayed in alphabetical order or sorted by salary. In this chapter, you will learn several sorting methods and techniques for comparing their performance. These techniques are useful not just for sorting algorithms, but also for analyzing other algorithms.

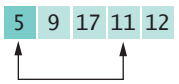
Once an array of elements is sorted, one can rapidly locate individual elements. You will study the *binary search* algorithm that carries out this fast lookup.

12.1 Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

A *sorting algorithm* rearranges the elements of an array so that they are stored in sorted order. In this section, we show you the first of several sorting algorithms, called **selection sort**. Consider the following array `a`:

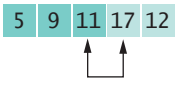
An obvious first step is to find the smallest element. In this case the smallest element is 5, stored in `a[3]`. You should move the 5 to the beginning of the array. Of course, there is already an element stored in `a[0]`, namely 11. Therefore you cannot simply move `a[3]` into `a[0]` without moving the 11 somewhere else. You don't yet know where the 11 should end up, but you know for certain that it should not be in `a[0]`. Simply get it out of the way by *swapping it* with `a[3]`:



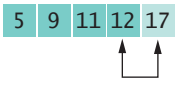
Now the first element is in the correct place. In the foregoing figure, the darker color indicates the portion of the array that is already sorted.

Next take the minimum of the remaining entries `a[1]`...`a[4]`. That minimum value, 9, is already in the correct place. You don't need to do anything in this case, simply extend the sorted area by one to the right:

Repeat the process. The minimum value of the unsorted region is 11, which needs to be swapped with the first value of the unsorted region, 17:



Now the unsorted region is only two elements long; keep to the same successful strategy. The minimum element is 12. Swap it with the first value, 17:



That leaves you with an unprocessed region of length 1, but of course a region of length 1 is always sorted. You are done.

If speed was not an issue for us, we could stop the discussion of sorting right here. However, the selection sort algorithm shows disappointing performance when run on large data sets, and it is worthwhile to study better sorting algorithms.

Here is the implementation of the selection sort algorithm:

ch12/selsort.cpp

```
1  #include <cstdlib>
2  #include <ctime>
3  #include <iostream>
4
5  using namespace std;
6
7  /**
8     Gets the position of the smallest element in an array range.
9     @param a the array
10    @param from the beginning of the range
11    @param to the end of the range
12    @return the position of the smallest element in
13    the range a[from]...a[to]
14 */
15 int min_position(int a[], int from, int to)
16 {
17    int min_pos = from;
18    for (int i = from + 1; i <= to; i++)
19    {
20       if (a[i] < a[min_pos]) { min_pos = i; }
21    }
22    return min_pos;
23 }
24
25 /**
26    Swaps two integers.
27    @param x the first integer to swap
28    @param y the second integer to swap
29 */
30 void swap(int& x, int& y)
31 {
32    int temp = x;
33    x = y;
34    y = temp;
35 }
36
37 /**
38    Sorts an array using the selection sort algorithm.
39    @param a the array to sort
40    @param size the number of elements in a
41 */
42 void selection_sort(int a[], int size)
43 {
44    int next; // The next position to be set to the minimum
45
46    for (next = 0; next < size - 1; next++)
47    {
48       // Find the position of the minimum
49       int min_pos = min_position(a, next, size - 1);
```

```
50          if (min_pos != next)
51          {
52             swap(a[min_pos], a[next]);
53          }
54       }
55    }
56
57    /**
58       Prints all elements in an array.
59       @param a the array to print
60       @param size the number of elements in a
61    */
62    void print(int a[], int size)
63    {
64       for (int i = 0; i < size; i++)
65       {
66          cout << a[i] << " ";
67       }
68       cout << endl;
69    }
70
71    int main()
72    {
73       srand(time(0));
74       const int SIZE = 20;
75       int values[SIZE];
76       for (int i = 0; i < SIZE; i++)
77       {
78          values[i] = rand() % 100;
79       }
80       print(values, SIZE);
81       selection_sort(values, SIZE);
82       print(values, SIZE);
83       return 0;
84    }
```

Program Run

```
60 47 70 39 6 12 96 93 83 53 36 29 50 97 94 95 38 17 8 26
6 8 12 17 26 29 36 38 39 47 50 53 60 70 83 93 94 95 96 97
```

1. What steps does the selection sort algorithm go through to sort the array 6 5 4 3 2 1?
2. How can you change the selection sort algorithm so that it sorts the elements in descending order (that is, with the largest element at the beginning of the array)?
3. Suppose we modified the selection sort algorithm to start at the end of the array, working toward the beginning. In each step, the current position is swapped with the minimum. What is the result of this modification?
4. Why do we need the `temp` variable in the `swap` function? What would happen if you simply assigned `a[i]` to `a[j]` and `a[j]` to `a[i]`?

Practice It Now you can try these exercises at the end of the chapter: R12.1, P12.1, P12.2.

12.2 Profiling the Selection Sort Algorithm

To measure the performance of a program, one could simply run it and measure how long it takes by using a stopwatch. However, most of our programs run very quickly, and it is not easy to time them accurately in this way. Furthermore, when a program does take a noticeable time to run, a certain amount of that time may simply be used for loading the program from disk into memory (for which it should not be penalized) or for screen output (whose speed depends on the computer model, even for computers with identical CPUs). Instead we use the `time` function. The call

```
int now = time(0);
```

sets `now` to the number of seconds that have elapsed since January 1, 1970. We don't care about this value, but if we have two time measurements, then their difference yields the elapsed time:

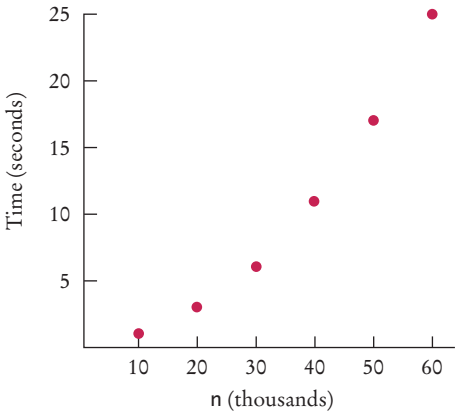
```
int before = time(0);
selection_sort(values, size);
int after = time(0);

cout << "Elapsed time = " << after - before
      << " seconds" << endl;
```

To measure the running time of a function, get the current time immediately before and after the function call.

By measuring the time just before and after the sorting, you don't count the time it takes to initialize the array or the time during which the program waits for the user to provide inputs. See `ch12/seltime.cpp` for the complete program. The table in Figure 1 shows the results of some sample runs.

These measurements were obtained on a Pentium processor with a clock speed of 1.67 GHz running Linux. On another computer, the actual numbers will differ, but the relationship between the numbers will be the same. Figure 1 shows a plot of the measurements.

As you can see, doubling the size of the data set more than doubles the time needed to sort it.



n	Seconds
10,000	1
20,000	3
30,000	6
40,000	11
50,000	17
60,000	25

Figure 1 Time Taken by Selection Sort

5. Approximately how many seconds would it take to sort a data set of 80,000 values?
6. Look at the graph in Figure 1. What mathematical shape does it resemble?

Practice It Now you can try these exercises at the end of the chapter: P12.3.

12.3 Analyzing the Performance of the Selection Sort Algorithm

Let us count the number of operations that a program must carry out to sort an array using the selection sort algorithm. Actually, we don't know how many machine operations are generated for each C++ instruction or which of those instructions are more time-consuming than others, but we can make a simplification. Simply count how often an element is *visited*. Each visit requires about the same amount of work by other operations, such as incrementing subscripts and comparing values.

Let n be the size of the array. First, you must find the smallest of n numbers. To achieve this, you must visit n elements. Then swap the elements, which takes two visits. (You may argue that there is a certain probability that you don't need to swap the values. That is true, and one can refine the computation to reflect that observation. As we will soon see, doing so would not affect the overall conclusion.) In the next step, you need to visit only $n - 1$ elements to find the minimum and then visit two of them to swap them. In the following step, $n - 2$ elements are visited to find the minimum. The last run visits two elements to find the minimum and requires two visits to swap the elements. Therefore, the total number of visits is

$$\begin{aligned} n + 2 + (n-1) + 2 + \cdots + 2 + 2 &= n + (n-1) + \cdots + 2 + (n-1) \cdot 2 \\ &= 2 + \cdots + (n-1) + n + (n-1) \cdot 2 \\ &= \frac{n(n+1)}{2} - 1 + (n-1) \cdot 2 \end{aligned}$$

because

$$1 + 2 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

After multiplying out and collecting terms of n, you find that the number of visits is

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

This is a quadratic equation in n. That explains why the graph of Figure 1 looks approximately like a parabola.

Now simplify the analysis further. When you plug in a large value for n (for example, 1,000 or 2,000), then $\frac{1}{2}n^2$ is 500,000 or 2,000,000. The lower term, $\frac{5}{2}n - 3$, doesn't contribute much at all; it is just 2,497 or 4,997, a drop in the bucket compared to the hundreds of thousands or even millions of comparisons specified by the $\frac{1}{2}n^2$ term.

Just ignore these lower-level terms. Next, ignore the constant factor $\frac{1}{2}$. You need not be interested in the actual count of visits for a single n. You need to compare the ratios of counts for different values of n^2. For example, you can say that sorting an array of 2,000 numbers requires four times as many visits as sorting an array of 1,000 numbers:

$$\frac{\left(\frac{1}{2} \cdot 2000^2\right)}{\left(\frac{1}{2} \cdot 1000^2\right)} = 4$$

The factor $\frac{1}{2}$ cancels out in comparisons of this kind. We will simply say, "The number of visits is of order n^2". That way, we can easily see that the number of comparisons increases fourfold when the size of the array doubles: $(2n)^2 = 4n^2$.

To indicate that the number of visits is of order n^2, computer scientists often use *big-Oh notation:* The number of visits is $O(n^2)$. This is a convenient shorthand.

To turn an exact expression such as

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

into big-Oh notation, simply locate the fastest-growing term, n^2, and ignore its constant coefficient, $\frac{1}{2}$ in this case, *no matter how large or small it may be.*

Computer scientists use big-Oh notation to describe how fast a function grows.

In general, the expression $f(n) = O(g(n))$ means that f grows no faster than g, or, more formally, that for all n larger than some threshold, the ratio $f(n)/g(n)$ is less than a constant value C. The function g is usually chosen to be very simple, such as n^2 in our example.

You observed before that the actual number of machine operations, and the actual amount of time that the computer spends on them, is approximately proportional to the number of element visits. Maybe there are about 10 machine operations (increments, comparisons, memory loads, and stores) for every element visit. The number of machine operations is then approximately $10 \times \frac{1}{2}n^2$. As before, we aren't interested in the coefficient, so we can say that the number of machine operations, and hence the time spent on the sorting, is of the order of n^2 or $O(n^2)$.

Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.

The sad fact remains that doubling the size of the array causes a fourfold increase in the time required for sorting it. When the size of an array increases by a factor of 100, the sorting time increases by a factor of 10,000. To sort an array of a million entries (for example, to create a telephone directory), takes 10,000 times as long as sorting 10,000 entries. If 10,000 entries can be sorted in about a second (as in our example), then sorting one million entries requires almost three hours. You will see in the next section how one can dramatically improve the performance of the sorting process by choosing a more sophisticated algorithm.

7. If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?
8. How large does n need to be so that $\frac{1}{2}n^2$ is bigger than $\frac{5}{2}n - 3$?
9. Section 6.2.7 has two algorithms for removing an element for an array of length n. How many array visits does each algorithm require on average?
10. Describe the number of array visits in Self Check 9 using the big-Oh notation.

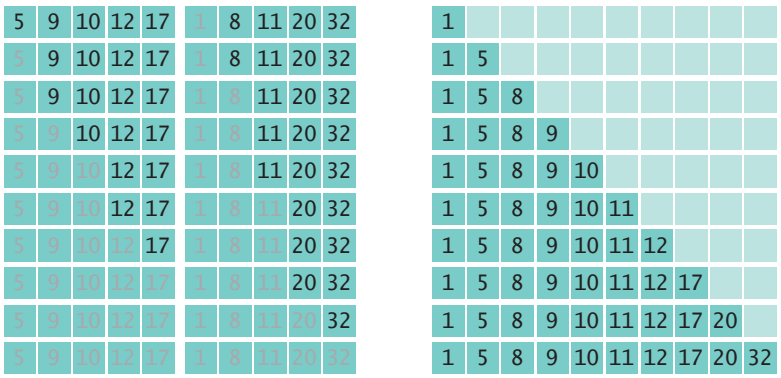
11. Consider this algorithm for sorting an array. Set k to the length of the array. Find the maximum of the first k elements. Remove it, using the second algorithm of Section 6.2.7. Decrement k and stop when it is 1. What is the algorithm's running time in big-Oh notation?

Practice It Now you can try these exercises at the end of the chapter: R12.3, R12.6, R12.9.

12.4 Merge Sort

In this section, you will learn about the merge sort algorithm, a much more efficient algorithm than selection sort. The basic idea behind merge sort is very simple. Suppose you have an array of 10 integers. Engage in a bit of wishful thinking and hope that the first half of the array is already perfectly sorted, and the second half is too, like this:

Now it is an easy matter to *merge* the two sorted arrays into a sorted array, simply by taking a new element from either the first or the second subarray and choosing the smaller of the elements each time:



In fact, you probably performed this merging before when you and a friend had to sort a pile of papers. You and the friend split the pile in half, each of you sorted your half, and then you merged the results together.

This is all well and good, but it doesn't seem to solve the problem for the computer. It still has to sort the first and second halves, because it can't very well ask a few buddies to pitch in. As it turns out, though, if the computer keeps dividing the array into smaller and smaller subarrays, sorting each half and merging them back together, it carries out dramatically fewer steps than the selection sort requires.

The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, then merging the sorted halves.

Let us write a program that implements this idea. Because we will call the `merge_sort` function multiple times to sort portions of the array, we will supply the range of elements that we would like to have sorted:

```
void merge_sort(int a[], int from, int to)
{
   if (from == to) { return; }
   int mid = (from + to) / 2;
```

```
   // Sort the first and the second half
   merge_sort(a, from, mid);
   merge_sort(a, mid + 1, to);
   merge(a, from, mid, to);
}
```

The `merge` function is somewhat long but quite straightforward—see the following code listing for details.

ch12/mergesort.cpp

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

/**
   Merges two adjacent ranges in an array.
   @param a the array with the elements to merge
   @param from the start of the first range
   @param mid the end of the first range
   @param to the end of the second range
*/
void merge(int a[], int from, int mid, int to)
{
   int n = to - from + 1; // Size of the range to be merged
   // Merge both halves into a temporary array b
   // We allocate the array dynamically because its size is only
   // known at run time—see Section 7.4
   int* b = new int[n];

   int i1 = from;
      // Next element to consider in the first half
   int i2 = mid + 1;
      // Next element to consider in the second half
   int j = 0; // Next open position in b

   // As long as neither i1 nor i2 is past the end, move the smaller
   // element into b

   while (i1 <= mid && i2 <= to)
   {
      if (a[i1] < a[i2])
      {
         b[j] = a[i1];
         i1++;
      }
      else
      {
         b[j] = a[i2];
         i2++;
      }
      j++;
   }

   // Note that only one of the two while loops below is executed

```

```
48       // Copy any remaining entries of the first half
49       while (i1 <= mid)
50       {
51          b[j] = a[i1];
52          i1++;
53          j++;
54       }
55       // Copy any remaining entries of the second half
56       while (i2 <= to)
57       {
58          b[j] = a[i2];
59          i2++;
60          j++;
61       }
62
63       // Copy back from the temporary array
64       for (j = 0; j < n; j++)
65       {
66          a[from + j] = b[j];
67       }
68
69       // The temporary array is no longer needed
70       delete[] b;
71    }
72
73    /**
74       Sorts the elements in a range of an array.
75       @param a the array with the elements to sort
76       @param from start of the range to sort
77       @param to end of the range to sort
78    */
79    void merge_sort(int a[], int from, int to)
80    {
81       if (from == to) { return; }
82       int mid = (from + to) / 2;
83       // Sort the first half and the second half
84       merge_sort(a, from, mid);
85       merge_sort(a, mid + 1, to);
86       merge(a, from, mid, to);
87    }
88
89    /**
90       Prints all elements in an array.
91       @param a the array to print
92       @param size the number of elements in a
93    */
94    void print(int a[], int size)
95    {
96       for (int i = 0; i < size; i++)
97       {
98          cout << a[i] << " ";
99       }
100      cout << endl;
101   }
102
103   int main()
104   {
105      srand(time(0));
106      const int SIZE = 20;
107      int values[SIZE];
```

```
108     for (int i = 0; i < SIZE; i++)
109     {
110        values[i] = rand() % 100;
111     }
112     print(values, SIZE);
113     merge_sort(values, 0, SIZE - 1);
114     print(values, SIZE);
115     return 0;
116  }
```

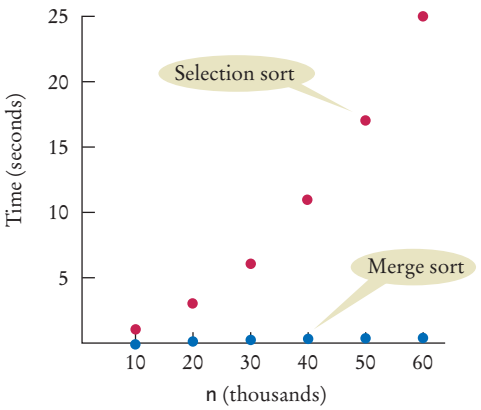
12. Why does only one of the two `while` loops at the end of the `merge` function do any work?
13. Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 1.
14. The merge sort algorithm processes an array by recursively processing two halves. Describe a similar recursive algorithm for computing the sum of all elements in an array.

Practice It Now you can try these exercises at the end of the chapter: R12.15, P12.4, P12.9.

12.5 Analyzing the Merge Sort Algorithm

The merge sort algorithm looks much more complicated than the selection sort algorithm, and it appears that it may well take much longer to carry out these repeated subdivisions. However, the timing results for merge sort look much better than those for selection sort (see `ch12/mergetime.cpp` and the table in Figure 2). Sorting an array with 60,000 elements takes less than one second on our test machine, whereas the selection sort takes 25 seconds.

In order to get precise timing results, it is best to run the algorithm multiple times, and then divide the total time by the number of runs. Figure 2 shows typical results and a graph plotting the relationship. Note that the graph does not have a parabolic shape. Instead, it appears as if the running time grows approximately linearly with the size of the array.



n	Merge Sort (seconds)	Selection Sort (seconds)
10,000	0.012	1
20,000	0.025	3
30,000	0.038	6
40,000	0.052	11
50,000	0.066	17
60,000	0.081	25

Figure 2 Merge Sort Timing versus Selection Sort

To understand why the merge sort algorithm is such a tremendous improvement, let us estimate the number of array element visits. First, we tackle the merge process that happens after the first and second halves have been sorted.

Each step in the merge process adds one more element to `b`. There are n elements in `b`. That element may come from the first or second half of `a`, and in most cases the elements from the two halves must be compared to see which one to take. Count that as 3 visits per element (one for `b` and one each for the two halves of `a`), or $3n$ visits total. Then you must copy back from `b` to `a`, yielding another $2n$ visits, for a total of $5n$.

If you let $T(n)$ denote the number of visits required to sort a range of n elements through the merge sort process, then you obtain

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 5n$$

because sorting each half takes $T(n/2)$ visits. Actually, if n is not even, then you have one array of size $(n - 1)/2$ and one of size $(n + 1)/2$. Although it turns out that this detail does not affect the outcome of the computation, you can assume for now that n is a power of 2, say $n = 2^m$. This way, all arrays can be evenly divided into two parts.

Unfortunately, the formula

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n$$

does not clearly tell you the relationship between n and $T(n)$. To understand the relationship, evaluate $T(n/2)$, using the same formula:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 5\frac{n}{2}$$

Therefore

$$T(n) = 2 \times 2T\left(\frac{n}{4}\right) + 5n + 5n$$

Do that again:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 5\frac{n}{4}$$

hence

$$T(n) = 2 \times 2 \times 2T\left(\frac{n}{8}\right) + 5n + 5n + 5n$$

This generalizes from 2, 4, 8, to arbitrary powers of 2:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 5nk$$

Recall that you assume that $n = 2^m$; hence, for $k = m$,

$$T(n) = 2^m T\left(\frac{n}{2^m}\right) + 5nm$$
$$= nT(1) + 5nm$$
$$= n + 5n\log_2(n)$$

Because $n = 2^m$, you have $m = \log_2(n)$.

To establish the growth order, you drop the lower order term n and are left with $5n\log_2(n)$. Drop the constant factor 5. It is also customary to drop the base of the logarithm because all logarithms are related by a constant factor. For example,

$$\log_2(x) = \log_{10}(x)/\log_{10}(2) \approx \log_{10}(x) \times 3.32193$$

Hence we say that merge sort is an $O(n\log(n))$ algorithm.

Merge sort is an $O(n\log(n))$ algorithm. The $n\log(n)$ function grows much more slowly than n^2.

Is the $O(n\log(n))$ merge sort algorithm better than an $O(n^2)$ selection sort algorithm? You bet it is. Recall that it took $100^2 = 10{,}000$ times as long to sort a million records as it took to sort 10,000 records with the $O(n^2)$ algorithm. With the $O(n\log(n))$ algorithm, the ratio is

$$\frac{1{,}000{,}000\log(1{,}000{,}000)}{10{,}000\log(10{,}000)} = 100\left(\frac{6}{4}\right) = 150$$

Suppose for the moment that merge sort takes the same time as selection sort to sort an array of 10,000 integers, that is, 1 second on the test machine. (Actually, as you have seen, it is much faster than that.) Then it would take about 150 seconds, or less than three minutes, to sort 1,000,000 integers. Contrast that with selection sort, which would take almost 3 hours for the same task. As you can see, even if it takes you several hours to learn about a better algorithm, that can be time well spent.

In this chapter you have barely begun to scratch the surface of this interesting topic. There are many sort algorithms, some with even better performance than the merge sort algorithm, and the analysis of these algorithms can be quite challenging. If you are a computer science major, you may revisit these important issues in later computer science classes.

SELF CHECK

15. Given the timing data for the merge sort algorithm in the table in Figure 2, how long would it take to sort an array of 100,000 values?
16. If you double the size of an array, how much longer will the merge sort algorithm take to sort the new array?

Practice It Now you can try these exercises at the end of the chapter: R12.7, R12.10, R12.11.

Special Topic 12.1

The Quicksort Algorithm

Quicksort is a commonly used algorithm that has the advantage over merge sort that no temporary arrays are required to sort and merge the partial results.

The quicksort algorithm, like merge sort, is based on the strategy of divide and conquer. To sort a range `a[from] . . . a[to]` of the array a, first rearrange the elements in the range so that no element in the range `a[from] . . . a[p]` is larger than any element in the range `a[p + 1] . . . a[to]`. This step is called *partitioning* the range.

For example, suppose we start with a range

Here is a partitioning of the range. Note that the partitions aren't yet sorted.

You'll see later how to obtain such a partition. In the next step, sort each partition, by recursively applying the same algorithm on the two partitions. That sorts the entire range, because the largest element in the first partition is at most as large as the smallest element in the second partition.

Quicksort is implemented recursively as follows:

```
void sort(int a[], int from, int to)
{
   if (from >= to) { return; }
   int p = partition(a, from, to);
   sort(a, from, p);
   sort(a, p + 1, to);
}
```

Let us return to the problem of partitioning a range. Pick an element from the range and call it the *pivot*. There are several variations of the quicksort algorithm. In the simplest one, we'll pick the first element of the range, `a[from]`, as the pivot.

Now form two regions `a[from] . . . a[i]`, consisting of values at most as large as the pivot and `a[j] . . . a[to]`, consisting of values at least as large as the pivot. The region `a[i + 1] . . . a[j - 1]` consists of values that haven't been analyzed yet. (See Figure 3.) At the beginning, both the left and right areas are empty; that is, `i = from - 1` and `j = to + 1`.

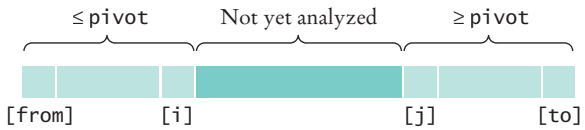


Figure 3 Partitioning a Range

Then keep incrementing `i` while `a[i] < pivot` and keep decrementing `j` while `a[j] > pivot`. Figure 4 shows `i` and `j` when that process stops.

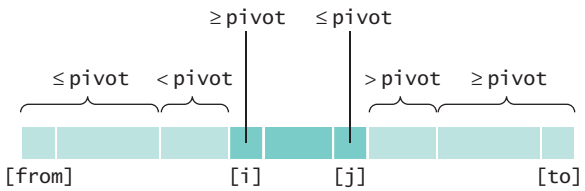


Figure 4 Extending the Partitions

Now swap the values in positions `i` and `j`, increasing both areas once more. Keep going while `i < j`. Here is the code for the `partition` function:

```
int partition(int a[], int from, int to)
{
   int pivot = a[from];
   int i = from - 1;
```

```
      int j = to + 1;
      while (i < j)
      {
         i++; while (a[i] < pivot) { i++; }
         j--; while (a[j] > pivot) { j--; }
         if (i < j) { swap(a[i], a[j]); }
      }
      return j;
   }
```

On average, the quicksort algorithm is an $O(n \log(n))$ algorithm. Because it is simpler, it runs faster than merge sort in most cases. There is just one unfortunate aspect to the quicksort algorithm. Its *worst-case* run-time behavior is $O(n^2)$. Moreover, if the pivot element is chosen as the first element of the region, that worst-case behavior occurs when the input set is already sorted—a common situation in practice. By selecting the pivot element more cleverly, we can make it extremely unlikely for the worst-case behavior to occur. Such "tuned" quicksort algorithms are commonly used because their performance is generally excellent. For example, the C library contains a function `qsort` that implements the quicksort algorithm.

12.6 Searching

Searching for an element in an array is an extremely common task. As with sorting, the right choice of algorithms can make a big difference.

12.6.1 Linear Search

Suppose you need to find the telephone number of your friend. If you have a telephone book, you can look up your friend's name quickly, because the telephone book is sorted alphabetically. However, now suppose you have a telephone number and you must know to whom it belongs (without actually calling the number). You could look through the telephone book, one number at a time, until you find the number. This would obviously be a tremendous amount of work.

This thought experiment shows the difference between a search through an unsorted data set and a search through a sorted data set.

A linear search examines all values in an array until it finds a match or reaches the end.

If you want to find a number in an array of values in arbitrary order, you must look through all elements until you have found a match or until you reach the end. This is called a *linear* or *sequential search*.

Here is a function that performs a linear search through an array of integers a for a given value (see `ch12/lsearch.cpp`). The function then returns the index of the match, or –1 if the value does not occur in `a`.

```
int linear_search(int a[], int size, int value)
{
   for (int i = 0; i < size; i++)
   {
      if (a[i] == value)
      {
         return i;
      }
   }
```

```
   return -1;
}
```

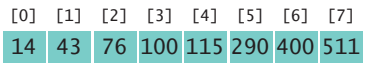
A linear search locates a value in an array in $O(n)$ steps.

How long does a linear search take? If you assume that the value is present in the array a, then the average search visits $n/2$ elements. If it is not present, then all n elements must be inspected to verify the absence. Either way, a linear search is an $O(n)$ algorithm.

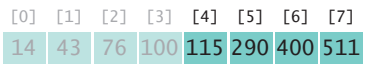
12.6.2 Binary Search

Now consider searching for an item in an array that has been previously sorted. Of course, you could still do a linear search, but it turns out you can do much better than that.

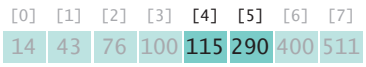
Here is a typical example. The data set is:



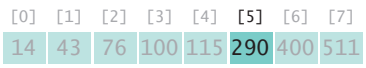
and you want to see whether the value 123 is in the data set. The last point in the first half of the data set, `a[3]`, is 100. It is smaller than the value you are looking for; hence, you should look in the second half of the data set for a match, that is, in the array



Now the last value of the first half of this array is 290; hence, the value must be located in the array



The last value of the first half of this very short array is 115, which is smaller than the value that you are searching, so you must look in the second half:



It is trivial to see that you don't have a match, because 123 ≠ 290. If you wanted to insert 123 into the array, you would need to insert it just before `a[5]`.

A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.

This search process is called a *binary search*, because the size of the search is cut in half in each step. That cutting in half works only because you know that the array of values is sorted.

The following function implements a binary search in a sorted array of integers (see `ch12/bsearch.cpp`). It returns the position of the match if the search succeeds, or –1 if the value is not found in the array:

```
int binary_search(int a[], int from, int to, int value)
{
   if (from > to)
   {
      return -1;
   }

   int mid = (from + to) / 2;
   if (a[mid] == value)
   {
      return mid;
   }
```

```
      else if (a[mid] < value)
      {
         return binary_search(a, mid + 1, to, value);
      }
      else
      {
         return binary_search(a, from, mid - 1, value);
      }
   }
```

Now determine the number of element visits required to carry out a search. Use the same technique as in the analysis of merge sort. Because you look at the middle element, which counts as one comparison, and then search either the left or the right array, you have

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Using the same equation,

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

By plugging this result into the original equation, you get

$$T(n) = T\left(\frac{n}{4}\right) + 2$$

This generalizes to

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

As in the analysis of merge sort, you make the simplifying assumption that n is a power of 2, $n = 2^m$, where $m = \log_2(n)$. Then you obtain

$$T(n) = 1 + \log_2(n)$$

Therefore, binary search is an $O(\log(n))$ algorithm.

This result makes intuitive sense. Suppose that n is 100. Then after each search, the size of the search range is cut in half, to 50, 25, 12, 6, 3, and 1. After seven comparisons we are done. This agrees with our formula, because $\log_2(100) \approx 6.64386$, and indeed the next larger power of 2 is $2^7 = 128$.

A binary search locates a value in a sorted array in $O(\log(n))$ steps.

Because a binary search is so much faster than a linear search, is it worthwhile to sort an array first and then use a binary search? It depends. If you only search the array once, then it is more efficient to pay for an $O(n)$ linear search than for an $O(n \log(n))$ sort and $O(\log(n))$ binary search. But if one makes a number of searches in the same array, then sorting it is definitely worthwhile.

17. Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?

18. What happens when you execute the binary search algorithm on an array that is not sorted?

19. Suppose a value is repeated in the array that the binary search algorithm searches. If you search for that value, which index is returned?
20. How can you modify the binary search algorithm to return the *lowest* index at which a (possibly repeated) value occurs? (*Hint:* If the range has length 1, the answer is easy. If `a[mid] < value`, then you know the lowest value must be in the upper half.)

Practice It Now you can try these exercises at the end of the chapter: R12.16, P12.12, P12.8.

Programming Tip 12.1

Library Functions for Sorting and Binary Search

If you need to sort or search values in your own programs, there is no need to implement your own algorithms. You can simply use functions in the C++ library. This note gives you a brief overview of the library functions for sorting and binary search. For more information on using library algorithms, see *Big C++, 2nd ed.*, by Cay Horstmann and Tim Budd (John Wiley & Sons, Inc., 2009).

You sort an array by calling the `sort` function with a pointer to the beginning and the end of the array:

```
sort(a, a + size);
```

Here *size* is the size of the array. For example,

```
int a[5] = { 60, 47, 70, 39, 6 };
sort(a, a + 5); // Now a contains 6, 39, 47, 60, 70
```

For a vector, the call looks slightly different:

```
sort(v.begin(), v.end());
```

The expressions `v.begin()` and `v.end()` are *iterators* that denote the beginning and ending positions of the vector. (As you will see in the next chapter, an iterator denotes a position in a container.)

If you have a sorted array or vector, you can use the library's `binary_search` function to test whether it contains a given value. For example, the call

```
binary_search(a, a + size, value)
```

returns `true` if the array a contains `value`. (Unlike our binary search function from Section 12.6, the library function does not return the position where the value was found.)

To search a vector, you call

```
binary_search(v.begin(), v.end(), value)
```

To use the `sort` or `binary_search` functions, you must include the `<algorithm>` header.

Special Topic 12.2

Defining an Ordering for Sorting Objects

When you use the `sort` library function, you must ensure that it is able to compare elements. Suppose that you want to sort an array of `Employee` objects. The compiler will complain that it does not know how to compare two employees.

There are several ways to overcome this problem. The simplest is to define the `<` operator for `Employee` objects:

```
bool operator<(const Employee& a, const Employee& b)
{
   return a.get_salary() < b.get_salary();
```

```
}
```

The curious name `operator<` indicates that this function defines a comparison operator. (See `ch12/stlsort/stlsort.cpp` for an example program.) For more information about defining your own operators, see *Big C++, 2nd ed.*, by Cay Horstmann and Tim Budd, Chapter 14 (John Wiley & Sons, Inc., 2009).

This `<` operator compares employees by salary. If you call `sort` to sort an array of employees, they will be sorted by increasing salary.

Random Fact 12.1 Cataloging Your Necktie Collection

People and companies use computers to organize just about every aspect of their lives. On the whole, computers are tremendously good for collecting and analyzing data. In fact, the power offered by computers and their software makes them seductive solutions for just about any organizational problem. It is easy to lose sight of the fact that using a computer is not always the best solution to a problem.

In 1983, the author John Bear wrote about a person who had come up with a novel use for the personal computers that had recently become available. That person cataloged his necktie collection, putting descriptions of the ties into a database and generating reports that listed them by color, price, or style. We can hope he had another use to justify the purchase of a piece of equipment worth several thousand dollars, but that particular application was so dear to his heart that he wanted the world to know about it. Perhaps not surprisingly, few other computer users shared that excitement, and you don't find the shelves of your local software store lined with necktie-cataloging software.

The phenomenon of using technology for its own sake is quite widespread. In the "Internet bubble" of 2000, hundreds of companies were founded on the premise that the Internet made it technologically possible to order items such as groceries and pet food from a home computer, and therefore the traditional stores would be replaced by web stores. However, technological feasibility did not ensure economic success. Trucking groceries and pet food to households was expensive, and few customers were willing to pay a premium for the added convenience.

Many elementary schools spend tremendous resources to bring computers and the Internet into the classroom. Indeed, it is easy to understand why teachers, school administrators, parents, politicians and equipment vendors are in favor of computers in classrooms. Isn't computer literacy absolutely essential for youngsters in the new millennium? Isn't it particularly important to give low-income kids, whose parents may not be able to afford a home computer, the opportunity to master computer skills? However, the total cost of running computers far exceeds the initial cost of the equipment. Some schools have had to make hard choices—should they lay off librarians and art instructors to hire more computer technicians, or should they let the equipment become useless? It is easy to get caught up in the technology hype without questioning whether the educational benefits justify the expense.

As computer programmers, we like to computerize everything. As computer professionals, though, we owe it to our employers and clients to understand which problems they want to solve and to deploy computers and software only where they add more value than cost.

CHAPTER SUMMARY

Describe the selection sort algorithm.

- The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

Measure the running time of a function.

- To measure the running time of a function, get the current time immediately before and after the function call.

Use the big-Oh notation to describe the running time of an algorithm.

- Computer scientists use big-Oh notation to describe how fast a function grows.
- Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.

Describe the merge sort algorithm.

- The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, then merging the sorted halves.

Contrast the running times of the merge sort and selection sort algorithms.

- Merge sort is an $O(n \log(n))$ algorithm. The $n \log(n)$ function grows much more slowly than n^2.

Describe the linear search and binary search algorithms and their running times.

- A linear search examines all values in an array until it finds a match or reaches the end.
- A linear search locates a value in an array in $O(n)$ steps.
- A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.
- A binary search locates a value in an array in $O(\log(n))$ steps.

REVIEW EXERCISES

R12.1 *Checking against off-by-one errors.* When writing the selection sort algorithm of Section 12.1, a programmer must make the usual choices of `<` against `<=`, `size` against `size - 1`, and `next` against `next + 1`. This is fertile ground for off-by-one errors. Make code walkthroughs of the algorithm with arrays of length 0, 1, 2, and 3 and check carefully that all index values are correct.

R12.2 What is the difference between searching and sorting?

R12.3 For the following expressions, what is the order of the growth of each?

a. $n^2 + 2n + 1$

b. $n^{10} + 9n^9 + 20n^8 + 145n^7$

c. $(n + 1)^4$

d. $(n^2 + n)^2$

e. $n + 0.001n^3$

f. $n^3 - 1000n^2 + 10^9$

g. $n + \log(n)$

h. $n^2 + n \log(n)$

i. $2^n + n^2$

j. $\dfrac{(n^3 + 2n)}{(n^2 + 0.75)}$

R12.4 You determined that the actual number of visits in the selection sort algorithm is

$$T(n) = \frac{1}{2}n^2 + \frac{5}{2}n - 3$$

You then characterized this function as having $O(n^2)$ growth. Compute the actual ratios

$$T(2,000)/T(1,000)$$
$$T(5,000)/T(1,000)$$
$$T(10,000)/T(1,000)$$

and compare them with

$$f(2,000)/f(1,000)$$
$$f(5,000)/f(1,000)$$
$$f(10,000)/f(1,000)$$

where $f(n) = n^2$.

R12.5 Suppose algorithm A takes five seconds to handle a data set of 1,000 records. If the algorithm A is an $O(n)$ algorithm, how long will it take to handle a data set of 2,000 records? Of 10,000 records?

R12.6 Suppose an algorithm takes five seconds to handle a data set of 1,000 records. Fill in the following table, which shows the approximate growth of the execution times depending on the complexity of the algorithm.

For example, because $3000^2 / 1000^2 = 9$, the $O(n^2)$ algorithm would take 9 times as long, or 45 seconds, to handle a data set of 3,000 records.

	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n \log(n))$	$O(2^n)$
1,000	5	5	5	5	5
2,000					
3,000		45			
10,000					

R12.7 Sort the following growth rates from slowest growth to fastest growth.

$O(n)$	$O(n \log(n))$
$O(n^3)$	$O(2^n)$
$O(n^n)$	$O(\sqrt{n})$
$O(\log(n))$	$O(n\sqrt{n})$
$O(n^2 \log(n))$	$O(n^{\log(n)})$

R12.8 What is the order of complexity of the standard algorithm to find the minimum value of an array? Of finding both the minimum and the maximum?

R12.9 What is the order of complexity of the following function?

```
int count(int a[], int size, int c)
{
   int count = 0;

   for (int i = 0; i < size; i++)
   {
      if (a[i] == c) { count++; }
   }
   return count;
}
```

R12.10 Your task is to remove all duplicates from an array. For example, if the array has the values

```
4   7   11   4   9   5   11   7   3   5
```

then the array should be changed to

```
4   7   11   9   5   3
```

Here is a simple algorithm. Look at `a[i]`. Count how many times it occurs in `a`. If the count is larger than 1, remove it. What is the order of complexity of this algorithm?

R12.11 Modify the merge sort algorithm to remove duplicates in the merging step to obtain an algorithm that removes duplicates from an array. Note that the resulting array does not have the same ordering as the original one. What is the efficiency of this algorithm?

R12.12 Develop an $O(n \log(n))$ algorithm for removing duplicates from an array if the resulting array must have the same ordering as the original array. When a value occurs multiple times, all but its first occurrence should be removed.

R12.13 Consider the following sorting algorithm. To sort an array `a`, make a second array `b` of the same size. Then insert elements from `a` into `b`, keeping `b` in sorted order. For each element, call the binary search function of Exercise P12.6 to determine where it needs to be inserted. To insert an element into the middle of an array, you need to move all elements above the insert location up.

Is this an efficient algorithm? Estimate the number of element visits in the sorting process. Assume that on average half of the elements of `b` need to be moved to insert a new element.

R12.14 Make a walkthrough of selection sort with the following data sets:

a. 4 7 11 4 9 5 11 7 3 5

b. –7 6 8 7 5 9 0 11 10 5 8

R12.15 Make a walkthrough of merge sort with the following data sets:

a. 5 11 7 3 5 4 7 11 4 9

b. 9 0 11 10 5 8 –7 6 8 7 5

R12.16 Make a walkthrough of the following:

a. Linear search for 7 in –7 1 3 3 4 7 11 13

b. Binary search for 8 in –7 2 2 3 4 7 8 11 13

c. Binary search for 8 in –7 1 2 3 5 7 10 13

PROGRAMMING EXERCISES

P12.1 Modify the selection sort algorithm to sort an array of strings by increasing length.

P12.2 Modify the selection sort algorithm to sort a vector of integers.

P12.3 Write a program that automatically generates the table of sample runs of the selection sort times as in Figure 1. The program should ask for the smallest and largest value of *n* and the number of measurements, then make all sample runs and display the table.

P12.4 Modify the merge sort algorithm to sort a vector of employees by salary.

P12.5 Write a telephone lookup program. Read a data set of 1,000 names and telephone numbers from a file that contains the numbers in random order. Handle lookups by name and also reverse lookups by phone number. Use a binary search for both lookups.

P12.6 Consider the binary search function in Section 12.6.2. If no match is found, the function returns –1. Modify the function so that it returns a `bool` value indicating whether a match was found. Add a reference parameter `pos`, which is set to the location of the match if the search was successful. If `a` was not found, set `pos` to the index of the next larger value instead, or to the array size if `a` is larger than all the elements of the array.

P12.7 Use the modification of the binary search function from Exercise P12.6 to sort an array. Make a second array of the same size as the array to be sorted. For each element in the first array, call binary search on the second array to find out where the new element should be inserted. Then move all elements above the insertion point up by one slot and insert the new element. Thus, the second array is always kept sorted. Implement this algorithm and measure its performance.

P12.8 Implement the `binary_search` function of Section 12.6.2 without recursion. *Hint:* While `from < to`, update either `from` or `to`, depending on which range should be searched.

P12.9 Implement the `merge_sort` function without recursion, where the size of the array is a power of 2. First merge adjacent regions of size 1, then adjacent regions of size 2, then adjacent regions of size 4, and so on.

P12.10 Implement the `merge_sort` function without recursion, where the size of the array is an arbitrary number. *Hint:* Keep merging adjacent areas whose size is a power of 2, and pay special attention to the last area in the array.

P12.11 Write a program that sorts a vector of `Employee` objects by the employee names and prints the results. Use the `sort` function from the C++ library.

P12.12 Write a program that keeps an appointment book. Make a class `Appointment` that stores a description of the appointment, the appointment day, the starting time, and the ending time. Your program should keep the appointments in a sorted vector. Users can add appointments and print out all appointments for a given day. When a new appointment is added, use binary search to find where it should be inserted in the vector. Do not add it if it conflicts with another appointment.

ANSWERS TO SELF-CHECK QUESTIONS

1. 1 | 5 4 3 2 6, 1 2 | 4 3 5 6, 1 2 3 4 5 6
2. In each step, find the *maximum* of the remaining elements and swap it with the current element (or see Self Check 3).
3. The modified algorithm sorts the array in descending order.
4. Dropping the `temp` variable would not work. Then `a[i]` and `a[j]` would end up being the same value.
5. Four times as long as 40,000 values, or about 50 seconds.
6. A parabola.
7. It takes about 100 times longer.
8. If n is 4, then $\frac{1}{2}n^2$ is 8 and $\frac{5}{2}n - 3$ is 7.
9. The first algorithm requires one visit, to store the new element. The second algorithm requires $T(p) = 2 \times (n - p - 1)$ visits, where p is the location at which the element is removed. We don't know where that element is, but if elements are removed at random locations, on average, half of the removals will be above the middle and half below, so we can assume an average p of $n / 2$ and $T(n) = 2 \times (n - n / 2 - 1) = n - 2$.
10. The first algorithm is $O(1)$, the second $O(n)$.
11. Let n be the length of the array. In the kth step, we need k visits to find the minimum. To remove it, we need an average of $k - 2$ visits (see Self Check 9). One additional visit is required to add t to the end. Thus, the kth step requires $2k - 1$ visits. Because k goes from n to 2, the total number of visits is

    $$\begin{aligned} &2n - 1 + 2(n - 1) - 1 + \ldots + 2 \cdot 3 - 1 + 2 \cdot 2 - 1 \\ &= 2(n + (n - 1) + \ldots + 3 + 2 + 1 - 1) - (n - 1) \\ &= n(n + 1) - 2 - n + 1 \text{ (because } 1 + 2 + 3 + \ldots + (n - 1) + n = n(n + 1)/2) \\ &= n^2 - 3 \end{aligned}$$

    Therefore, the total number of visits is $O(n^2)$.
12. When the preceding `while` loop ends, the loop condition must be false, that is, `i1 > mid` (in which case the first loop isn't executed), or `i2 > to` (in which case the second loop isn't executed).
13. First sort 8 7 6 5. Recursively, first sort 8 7. Recursively, first sort 8. It's sorted. Sort 7. It's sorted. Merge them: 7 8. Do the same with 6 5 to get 5 6. Merge them to 5 6 7 8. Do the same with 4 3 2 1: Sort 4 3 by sorting 4 and 3 and merging them to 3 4. Sort 2 1 by sorting 2 and 1 and merging them to 1 2. Merge 3 4 and 1 2 to 1 2 3 4. Finally, merge 5 6 7 8 and 1 2 3 4 to 1 2 3 4 5 6 7 8.
14. If the array size is 1, return its only element as the sum. Otherwise, recursively compute the sum of the first and second subarray and return the sum of these two values.
15. Approximately $100{,}000 \cdot \log(100{,}000) / 50{,}000 \cdot \log(50{,}000) = 2 \cdot 5 / 4.7 = 2.13$ times the time required for 50,000 values. That's $2.13 \cdot 97$ milliseconds or approximately 207 milliseconds.
16. $\dfrac{2n\log(2n)}{n\log(n)} = 2\dfrac{(1 + \log(2))}{\log(n)}$ For $n > 2$, this is a value < 3.
17. On average, you'd make 500,000 comparisons.

18. The algorithm may falsely report that an element is not present. For example, consider the task of finding 9 in the array 17 5 9 8 1 32 20 12. Because `a[3] < 9`, the algorithm will look at the second half of the array 1 32 20 12, which does not contain the value 9.

19. Any of the indexes at which the element exists may be returned. For example, if the array is 1 4 5 5 5 7 8 8, and you search for 5, then `a[3]` is visited first, and a position of 3 is returned.

20.
```
int binary_search(int a[], int from, int to, int value)
{
   if (from == to)  // Range has length 1
   {
      if (a[from] == value) { return from; }
      else { return -1; }
   }

   int mid = (from + to) / 2;
   if (a[mid] < value) // Value must be in the upper half
   {
      return binary_search(a, mid + 1, to, value);
   }
   else // Keep on searching in the lower half
   {
      return binary_search(a, from, mid, value);
   }
}
```