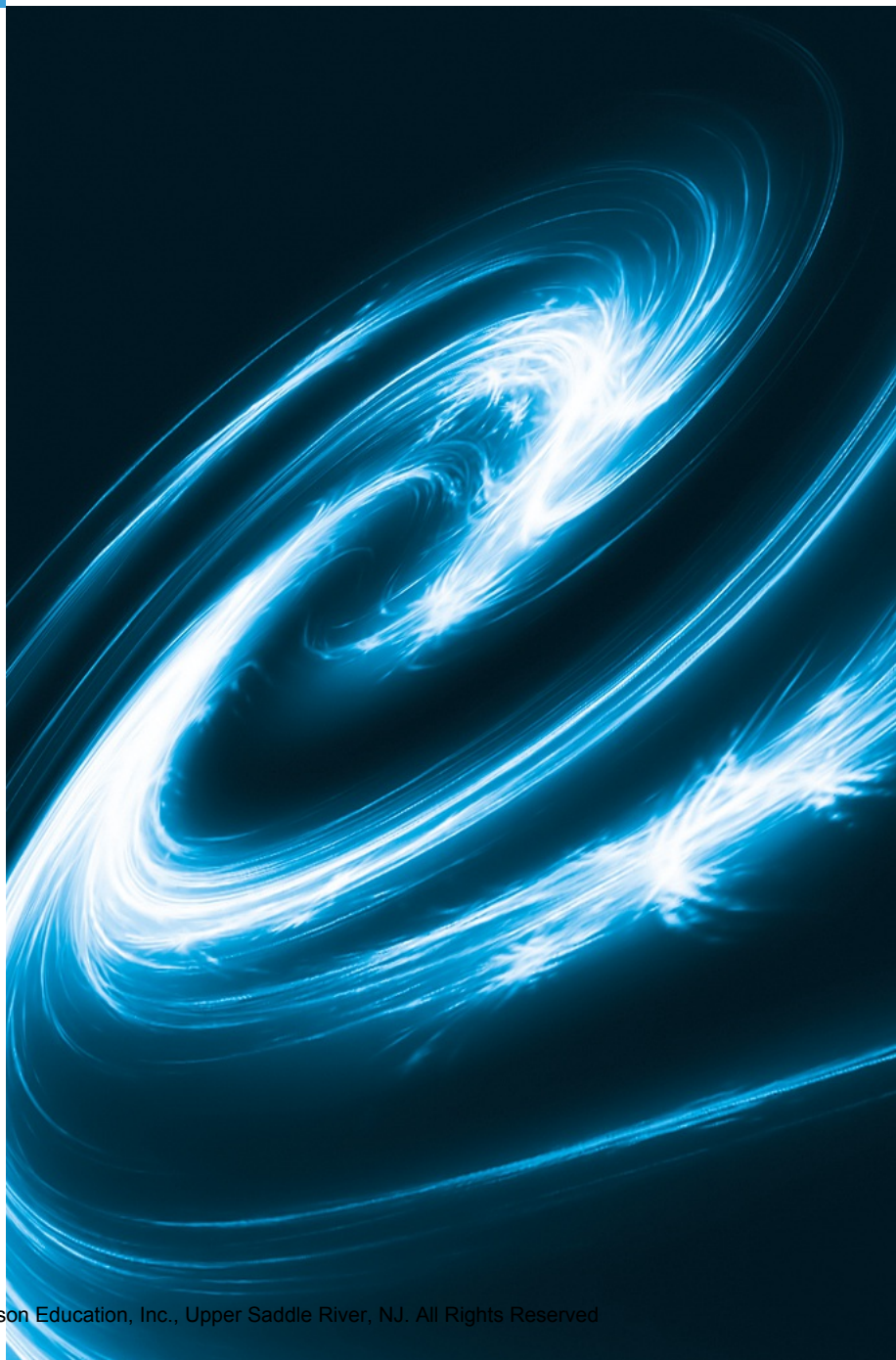
E

Sorting: A Deeper Look



With sobs and tears
he sorted out
Those of the largest size …
—Lewis Carroll

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
—William Shakespeare

It is an immutable law in business that words are words, explanations are explanations, promises are promises — but only performance is reality.
—Harold S. Green

Objectives

In this appendix, you'll learn:

- To sort an array using the selection sort algorithm.
- To sort an array using the insertion sort algorithm.
- To sort an array using the recursive merge sort algorithm.
- To determine the efficiency of searching and sorting algorithms and express it in "Big O" notation.
- To explore (in the exercises) additional recursive sorts, including quicksort and a recursive selection sort.
- To explore (in the exercises) the high performance bucket sort.

Outline

E.1 Introduction

As you learned in Chapter 6, sorting places data in order, typically ascending or descending, based on one or more sort keys. This appendix introduces the selection sort and insertion sort algorithms, along with the more efficient, but more complex, merge sort. We introduce **Big O notation**, which is used to estimate the worst-case run time for an algorithm—that is, how hard an algorithm may have to work to solve a problem.

An important point to understand about sorting is that the end result—the sorted array of data—will be the same no matter which sorting algorithm you use. The choice of algorithm affects only the run time and memory use of the program. The first two sorting algorithms we study here—selection sort and insertion sort—are easy to program, but inefficient. The third algorithm—recursive merge sort—is more efficient than selection sort and insertion sort, but harder to program.

The exercises present two more recursive sorts—quicksort and a recursive version of selection sort. Another exercise presents the bucket sort, which achieves high performance by clever use of considerably more memory than the other sorts we discuss.

E.2 Big O Notation

Suppose an algorithm is designed to test whether the first element of an array is equal to the second element of the array. If the array has 10 elements, this algorithm requires one comparison. If the array has 1000 elements, the algorithm still requires one comparison. In fact, the algorithm is completely independent of the number of elements in the array. This algorithm is said to have a **constant run time**, which is represented in Big O notation as ***O*(1)** and pronounced "order 1." An algorithm that is $O(1)$ does not necessarily require only one comparison. $O(1)$ just means that the number of comparisons is *constant*—it does not grow as the size of the array increases. An algorithm that tests whether the first element of an array is equal to any of the next three elements is still $O(1)$ even though it requires three comparisons.

An algorithm that tests whether the first element of an array is equal to *any* of the other elements of the array will require at most $n - 1$ comparisons, where n is the number of elements in the array. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1000 elements, this algorithm requires up to 999 comparisons. As n grows larger, the n part of the expression "dominates," and subtracting one becomes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows. For this reason, an algorithm that requires a total of $n - 1$ comparisons (such as the one we described earlier) is said to be ***O*(*n*)**. An $O(n)$ algorithm is referred to as having a **linear run time**. $O(n)$ is often pronounced "on the order of n" or more simply "order n."

Suppose you have an algorithm that tests whether *any* element of an array is duplicated elsewhere in the array. The first element must be compared with every other element in the array. The second element must be compared with every other element except the first—it was already compared to the first. The third element must be compared with every other element except the first two. In the end, this algorithm will end up making $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Again, Big O notation highlights the n^2 term, leaving $n^2/2$. But as we'll soon see, constant factors are omitted in Big O notation.

Big O is concerned with how an algorithm's run time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, the algorithm will require 64 comparisons. With this algorithm, doubling the number of elements quadruples the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, the algorithm will require 32 comparisons. Again, doubling the number of elements quadruples the number of comparisons. Both of these algorithms grow as the square of n, so Big O ignores the constant and both algorithms are considered to be **$O(n^2)$**, which is referred to as **quadratic run time** and pronounced "on the order of n-squared" or more simply "order n-squared."

When n is small, $O(n^2)$ algorithms (running on today's billion-operation-per-second personal computers) will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation. An $O(n^2)$ algorithm running on a million-element array would require a trillion "operations" (where each could actually require several machine instructions to execute). This could require a few hours to execute. A billion-element array would require a quintillion operations, a number so large that the algorithm could take decades! $O(n^2)$ algorithms, unfortunately, are easy to write, as you'll see in this appendix. You'll also see an algorithm with a more favorable Big O measure. Efficient algorithms often take a bit more cleverness and work to create, but their superior performance can be well worth the extra effort, especially as n gets large and as algorithms are combined into larger programs.

E.3 Selection Sort

Selection sort is a simple, but inefficient, sorting algorithm. The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest of those remaining) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last, leaving the largest element as the last. After the ith iteration, the smallest i positions of the array will be sorted into increasing order in the first i positions of the array.

As an example, consider the array

```
34   56   4   10   77   51   93   30   5   52
```

A program that implements selection sort first determines the smallest element (4) of this array which is contained in the third element of the array (i.e., element 2 because array subscripts start at 0). The program swaps 4 with 34, resulting in

```
4   56   34   10   77   51   93   30   5   52
```

The program then determines the smallest of the remaining elements (all elements except 4), which is 5, contained at array subscript 8. The program swaps 5 with 56, resulting in

```
4   5   34   10   77   51   93   30   56   52
```

On the third iteration, the program determines the next smallest value (10) and swaps it with 34.

```
4   5   10   34   77   51   93   30   56   52
```

The process continues until after nine iterations the array is fully sorted.

```
4   5   10   30   34   51   52   56   77   93
```

After the first iteration, the smallest element is in the first position. After the second iteration, the two smallest elements are in order in the first two positions. After the third iteration, the three smallest elements are in order in the first three positions.

Figure E.1 implements the selection sort algorithm on the array `array`, which is initialized with 10 random `int`s (possibly duplicates). The `main` function prints the unsorted array, calls the function `sort` on the array, and then prints the array again after it has been sorted.

```
1  /* Fig. E.1: figE_01.c
2     The selection sort algorithm. */
3  #define SIZE 10
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  /* function prototypes */
9  void selectionSort( int array[], int length );
10 void swap( int array[], int first, int second );
11 void printPass( int array[], int length, int pass, int index );
12
13 int main( void )
14 {
15    int array[ SIZE ]; /* declare the array of ints to be sorted */
16    int i; /* int used in for loop */
17
18    srand( time( NULL ) ); /* seed the rand function */
19
20    for ( i = 0; i < SIZE; i++ )
21       array[ i ] = rand() % 90 + 10; /* give each element a value */
22
23    puts( "Unsorted array:" );
24
25    for ( i = 0; i < SIZE; i++ ) /* print the array */
26       printf( "%d  ", array[ i ] );
27
28    puts( "\n" );
29    selectionSort( array, SIZE );
30    puts( "Sorted array:" );
31
```

Fig. E.1 | Selection sort algorithm. (Part 1 of 3.)

```
32     for ( i = 0; i < SIZE; i++ ) /* print the array */
33        printf( "%d  ", array[ i ] );
34
35  } /* end function main */
36
37  /* function that selection sorts the array */
38  void selectionSort( int array[], int length )
39  {
40     int smallest; /* index of smallest element */
41     int i, j; /* ints used in for loops */
42
43     /* loop over length - 1 elements */
44     for ( i = 0; i < length - 1; i++ ) {
45        smallest = i; /* first index of remaining array */
46
47         /* loop to find index of smallest element */
48         for ( j = i + 1; j < length; j++ )
49            if ( array[ j ] < array[ smallest ] )
50               smallest = j;
51
52         swap( array, i, smallest ); /* swap smallest element */
53         printPass( array, length, i + 1, smallest ); /* output pass */
54     } /* end for */
55  } /* end function selectionSort */
56
57  /* function that swaps two elements in the array */
58  void swap( int array[], int first, int second )
59  {
60     int temp; /* temporary integer */
61     temp = array[ first ];
62     array[ first ] = array[ second ];
63     array[ second ] = temp;
64  } /* end function swap */
65
66  /* function that prints a pass of the algorithm */
67  void printPass( int array[], int length, int pass, int index )
68  {
69     int i; /* int used in for loop */
70
71     printf( "After pass %2d: ", pass );
72
73     /* output elements till selected item */
74     for ( i = 0; i < index; i++ )
75        printf( "%d  ", array[ i ] );
76
77     printf( "%d* ", array[ index ] ); /* indicate swap */
78
79     /* finish outputting array */
80     for ( i = index + 1; i < length; i++ )
81        printf( "%d  ", array[ i ] );
82
83     printf( "%s", "\n               " ); /* for alignment */
```

Fig. E.1 | Selection sort algorithm. (Part 2 of 3.)

```
84
85      /* indicate amount of array that is sorted */
86      for ( i = 0; i < pass; i++ )
87         printf( "%s", "--  " );
88
89      puts( "\n" ); /* add newline */
90   } /* end function printPass */
```

```
Unsorted array:
72  34  88  14  32  12  34  77  56  83
After pass  1: 12  34  88  14  32  72* 34  77  56  83
               --
After pass  2: 12  14  88  34* 32  72  34  77  56  83
               --  --
After pass  3: 12  14  32  34  88* 72  34  77  56  83
               --  --  --
After pass  4: 12  14  32  34* 88  72  34  77  56  83
               --  --  --  --
After pass  5: 12  14  32  34  34  72  88* 77  56  83
               --  --  --  --  --
After pass  6: 12  14  32  34  34  56  88  77  72* 83
               --  --  --  --  --  --
After pass  7: 12  14  32  34  34  56  72  77  88* 83
               --  --  --  --  --  --  --
After pass  8: 12  14  32  34  34  56  72  77* 88  83
               --  --  --  --  --  --  --  --
After pass  9: 12  14  32  34  34  56  72  77  83  88*
               --  --  --  --  --  --  --  --  --
After pass 10: 12  14  32  34  34  56  72  77  83  88*
               --  --  --  --  --  --  --  --  --  --
Sorted array:
12  14  32  34  34  56  72  77  83  88
```

Fig. E.1 | Selection sort algorithm. (Part 3 of 3.)

Lines 39–56 define the `selectionSort` function. Line 41 declares the variable `smallest`, which stores the index of the smallest element in the remaining array. Lines 45–55 loop `SIZE - 1` times. Line 46 assigns the index of the smallest element to the current item. Lines 49–51 loop over the remaining elements in the array. For each of these elements, line 50 compares its value to the value of the smallest element. If the current element is smaller than the smallest element, line 51 assigns the current element's index to `smallest`. When this loop finishes, `smallest` contains the index of the smallest element in the remaining array. Line 53 calls function `swap` (lines 59–65) to place the smallest remaining element in the next spot in the array.

The output of this program uses dashes to indicate the portion of the array that is guaranteed to be sorted after each pass. An asterisk is placed next to the position of the element that was swapped with the smallest element on that pass. On each pass, the element to the left of the asterisk and the element above the rightmost set of dashes were the two values that were swapped.

Efficiency of Selection Sort

The selection sort algorithm runs in $O(n^2)$ time. The `selectionSort` method in lines 39–56 of Fig. E.1, which implements the selection sort algorithm, contains two `for` loops. The

outer `for` loop (lines 45–55) iterates over the first $n - 1$ elements in the array, swapping the smallest remaining item into its sorted position. The inner `for` loop (lines 49–51) iterates over each item in the remaining array, searching for the smallest element. This loop executes $n - 1$ times during the first iteration of the outer loop, $n - 2$ times during the second iteration, then $n - 3, \ldots, 3, 2, 1$. This inner loop iterates a total of $n(n - 1) / 2$ or $(n^2 - n)/2$. In Big O notation, smaller terms drop out and constants are ignored, leaving a Big O of $O(n^2)$.

E.4 Insertion Sort

Insertion sort is another simple, but inefficient, sorting algorithm. The first iteration of this algorithm takes the second element in the array and, if it's less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the ith iteration of this algorithm, the first i elements in the original array will be sorted.

Consider as an example the following array [*Note:* This array is identical to the array used in the discussions of selection sort and merge sort.]

```
34   56   4   10   77   51   93   30   5   52
```

A program that implements the insertion sort algorithm will first look at the first two elements of the array, 34 and 56. These two elements are already in order, so the program continues (if they were out of order, the program would swap them).

In the next iteration, the program looks at the third value, 4. This value is less than 56, so the program stores 4 in a temporary variable and moves 56 one element to the right. The program then checks and determines that 4 is less than 34, so it moves 34 one element to the right. The program has now reached the beginning of the array, so it places 4 in element 0. The array now is

```
4   34   56   10   77   51   93   30   5   52
```

In the next iteration, the program stores the value 10 in a temporary variable. Then the program compares 10 to 56 and moves 56 one element to the right because it's larger than 10. The program then compares 10 to 34, moving 34 right one element. When the program compares 10 to 4, it observes that 10 is larger than 4 and places 10 in element 1. The array now is

```
4   10   34   56   77   51   93   30   5   52
```

Using this algorithm, at the ith iteration, the first $i + 1$ elements of the original array are sorted with respect to one another. They may not be in their final locations, however, because smaller values may be located later in the array.

Figure E.2 implements the insertion sort algorithm. Lines 38–58 declare the `insertionSort` function. Line 40 declares the variable `insert`, which holds the element you're going to insert while you move the other elements. Lines 44–57 loop over `SIZE - 1` items in the array. In each iteration, line 46 stores in `insert` the value of the element that will be inserted into the sorted portion of the array. Line 45 declares and initializes the variable `moveItem`, which keeps track of where to insert the element. Lines 49–53 loop to locate the correct position where the element should be inserted. The loop terminates either when the program reaches the front of the array or when it reaches an element that is less than the value to be inserted. Line 51 moves an element to the right, and line 52 decre-

ments the position at which to insert the next element. After the loop ends, line 55 inserts the element into place. The output of this program uses dashes to indicate the portion of the array that is sorted after each pass. An asterisk is placed next to the element that was inserted into place on that pass.

```
1  /* Fig. E.2: figE_02.c
2     The insertion sort algorithm. */
3  #define SIZE 10
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  /* function prototypes */
9  void insertionSort( int array[], int length );
10 void printPass( int array[], int length, int pass, int index );
11
12 int main( void )
13 {
14    int array[ SIZE ]; /* declare the array of ints to be sorted */
15    int i; /* int used in for loop */
16
17    srand( time( NULL ) ); /* seed the rand function */
18
19    for ( i = 0; i < SIZE; i++ )
20       array[ i ] = rand() % 90 + 10; /* give each element a value */
21
22    puts( "Unsorted array:" );
23
24    for ( i = 0; i < SIZE; i++ ) /* print the array */
25       printf( "%d  ", array[ i ] );
26
27    puts( "\n" );
28    insertionSort( array, SIZE );
29    puts( "Sorted array:" );
30
31    for ( i = 0; i < SIZE; i++ ) /* print the array */
32       printf( "%d  ", array[ i ] );
33
34 } /* end function main */
35
36 /* function that sorts the array */
37 void insertionSort( int array[], int length )
38 {
39    int insert; /* temporary variable to hold element to insert */
40    int i; /* int used in for loop */
41
42    /* loop over length - 1 elements */
43    for ( i = 1; i < length; i++ ) {
44        int moveItem = i; /* initialize location to place element */
45        insert = array[ i ];
46
```

Fig. E.2 | Insertion sort algorithm. (Part 1 of 3.)

```
47          /* search for place to put current element */
48          while ( moveItem > 0 && array[ moveItem - 1 ] > insert ) {
49              /* shift element right one slot */
50              array[ moveItem ] = array[ moveItem - 1 ];
51              --moveItem;
52          } /* end while */
53
54          array[ moveItem ] = insert; /* place inserted element */
55          printPass( array, length, i, moveItem );
56      } /* end for */
57  } /* end function insertionSort */
58
59  /* function that prints a pass of the algorithm */
60  void printPass( int array[], int length, int pass, int index )
61  {
62      int i; /* int used in for loop */
63
64      printf( "After pass %2d: ", pass );
65
66      /* output elements till selected item */
67      for ( i = 0; i < index; i++ )
68          printf( "%d  ", array[ i ] );
69
70      printf( "%d* ", array[ index ] ); /* indicate swap */
71
72      /* finish outputting array */
73      for ( i = index + 1; i < length; i++ )
74          printf( "%d  ", array[ i ] );
75
76      puts( "%s* ", "\n               " ); /* for alignment */
77
78      /* indicate amount of array that is sorted */
79      for ( i = 0; i <= pass; i++ )
80          printf( "%d* ", "--  " );
81
82      puts( "" ); /* add newline */
83  } /* end function printPass */
```

```
Unsorted array:
72  16  11  92  63  99  59  82  99  30

After pass  1: 16* 72  11  92  63  99  59  82  99  30
               --  --
After pass  2: 11* 16  72  92  63  99  59  82  99  30
               --  --  --
After pass  3: 11  16  72  92* 63  99  59  82  99  30
               --  --  --  --
After pass  4: 11  16  63* 72  92  99  59  82  99  30
               --  --  --  --  --
After pass  5: 11  16  63  72  92  99* 59  82  99  30
               --  --  --  --  --  --
```

Fig. E.2 | Insertion sort algorithm. (Part 2 of 3.)

```
After pass  5: 11  16  63  72  92  99* 59  82  99  30
               --  --  --  --  --  --
After pass  6: 11  16  59* 63  72  92  99  82  99  30
               --  --  --  --  --  --  --
After pass  7: 11  16  59  63  72  82* 92  99  99  30
               --  --  --  --  --  --  --  --
After pass  8: 11  16  59  63  72  82  92  99  99* 30
               --  --  --  --  --  --  --  --  --
After pass  9: 11  16  30* 59  63  72  82  92  99  99
               --  --  --  --  --  --  --  --  --  --
Sorted array:
11  16  30  59  63  72  82  92  99  99
```

Fig. E.2 | Insertion sort algorithm. (Part 3 of 3.)

Efficiency of Insertion Sort

The insertion sort algorithm also runs in $O(n^2)$ time. Like selection sort, the `insertionSort` function (lines 38–58) uses two loops. The `for` loop (lines 44–57) iterates `SIZE - 1` times, inserting an element into the appropriate position in the elements sorted so far. For the purposes of this application, `SIZE - 1` is equivalent to $n - 1$ (as `SIZE` is the size of the array). The `while` loop (lines 49–53) iterates over the preceding elements in the array. In the worst case, this `while` loop requires $n - 1$ comparisons. Each individual loop runs in $O(n)$ time. In Big O notation, nested loops mean that you must multiply the number of iterations of each loop. For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each $O(n)$ iterations of the outer loop, there will be $O(n)$ iterations of the inner loop. Multiplying these values results in a Big O of $O(n^2)$.

E.5 Merge Sort

Merge sort is an efficient sorting algorithm, but is conceptually more complex than selection sort and insertion sort. The merge sort algorithm sorts an array by splitting it into two equal-sized subarrays, sorting each subarray, then merging them into one larger array. With an odd number of elements, the algorithm creates the two subarrays such that one has one more element than the other.

The implementation of merge sort in this example is recursive. The base case is an array with one element. A one-element array is, of course, sorted, so merge sort immediately returns when it's called with a one-element array. The recursion step splits an array of two or more elements into two equal-sized subarrays, recursively sorts each subarray, then merges them into one larger, sorted array. [Again, if there is an odd number of elements, one subarray is one element larger than the other.]

Suppose the algorithm has already merged smaller arrays to create sorted arrays A:

```
   4   10   34   56   77
```

and B:

```
   5   30   51   52   93
```

Merge sort combines these two arrays into one larger, sorted array. The smallest element in A is 4 (located in the element zero of A). The smallest element in B is 5 (located in the

zeroth index of B). To determine the smallest element in the larger array, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the first element in the merged array. The algorithm continues by comparing 10 (the second element in A) to 5 (the first element in B). The value from B is smaller, so 5 becomes the second element in the larger array. The algorithm continues by comparing 10 to 30, with 10 becoming the third element in the array, and so on.

Figure E.3 implements the merge sort algorithm, and lines 35–38 define the `mergeSort` function. Line 37 calls function `sortSubArray` with `0` and `SIZE - 1` as the arguments. The arguments correspond to the beginning and ending indices of the array to be sorted, causing `sortSubArray` to operate on the entire array. Function `sortSubArray` is defined in lines 41–66. Line 46 tests the base case. If the size of the array is `1`, the array is sorted, so the function simply returns immediately. If the size of the array is greater than `1`, the function splits the array in two, recursively calls function `sortSubArray` to sort the two subarrays, then merges them. Line 60 recursively calls function `sortSubArray` on the first half of the array, and line 61 recursively calls function `sortSubArray` on the second half of the array. When these two function calls return, each half of the array has been sorted. Line 64 calls function `merge` (lines 69–111) on the two halves of the array to combine the two sorted arrays into one larger sorted array.

```
1  /* Fig. E.3: figE_03.c
2     The merge sort algorithm. */
3  #define SIZE 10
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  /* function prototypes */
9  void mergeSort( int array[], int length );
10 void sortSubArray( int array[], int low, int high );
11 void merge( int array[], int left, int middle1, int middle2, int right );
12 void displayElements( int array[], int length );
13 void displaySubArray( int array[], int left, int right );
14
15 int main( void )
16 {
17    int array[ SIZE ]; /* declare the array of ints to be sorted */
18    int i; /* int used in for loop */
19
20    srand( time( NULL ) ); /* seed the rand function */
21
22    for ( i = 0; i < SIZE; i++ )
23       array[ i ] = rand() % 90 + 10; /* give each element a value */
24
25    puts( "Unsorted array:" );
26    displayElements( array, SIZE ); /* print the array */
27    puts( "\n" );
28    mergeSort( array, SIZE ); /* merge sort the array */
29    puts( "Sorted array:" );
30    displayElements( array, SIZE ); /* print the array */
```

Fig. E.3 | Merge sort algorithm. (Part 1 of 5.)

```
31  } /* end function main */
32
33  /* function that merge sorts the array */
34  void mergeSort( int array[], int length )
35  {
36     sortSubArray( array, 0, length - 1 );
37  } /* end function mergeSort */
38
39  /* function that sorts a piece of the array */
40  void sortSubArray( int array[], int low, int high )
41  {
42     int middle1, middle2; /* ints that record where the array is split */
43
44     /* test base case: size of array is 1 */
45     if ( ( high - low ) >= 1 ) { /* if not base case... */
46        middle1 = ( low + high ) / 2;
47        middle2 = middle1 + 1;
48
49        /* output split step */
50        printf( "%s* ", "split:   " );
51        displaySubArray( array, low, high );
52        printf( "%s* ", "\n         " );
53        displaySubArray( array, low, middle1 );
54        printf( "%s* ", "\n         " );
55        displaySubArray( array, middle2, high );
56        puts( "\n" );
57
58        /* split array in half and sort each half recursively */
59        sortSubArray( array, low, middle1 ); /* first half */
60        sortSubArray( array, middle2, high ); /* second half */
61
62        /* merge the two sorted arrays */
63        merge( array, low, middle1, middle2, high );
64     } /* end if */
65  } /* end function sortSubArray */
66
67  /* merge two sorted subarrays into one sorted subarray */
68  void merge( int array[], int left, int middle1, int middle2, int right )
69  {
70     int leftIndex = left; /* index into left subarray */
71     int rightIndex = middle2; /* index into right subarray */
72     int combinedIndex = left; /* index into temporary array */
73     int tempArray[ SIZE ]; /* temporary array */
74     int i; /* int used in for loop */
75
76     /* output two subarrays before merging */
77     printf( "%s* ", "merge:   " );
78     displaySubArray( array, left, middle1 );
79     printf( "%s* ", "\n         " );
80     displaySubArray( array, middle2, right );
81     puts( "" );
```

Fig. E.3 | Merge sort algorithm. (Part 2 of 5.)

```
82
83     /* merge the subarrays until the end of one is reached */
84     while ( leftIndex <= middle1 && rightIndex <= right ) {
85        /* place the smaller of the two current elements in result */
86        /* and move to the next space in the subarray */
87        if ( array[ leftIndex ] <= array[ rightIndex ] )
88           tempArray[ combinedIndex++ ] = array[ leftIndex++ ];
89        else
90           tempArray[ combinedIndex++ ] = array[ rightIndex++ ];
91     } /* end while */
92
93     if ( leftIndex == middle2 ) { /* if at end of left subarray ... */
94        while ( rightIndex <= right ) /* copy the right subarray */
95           tempArray[ combinedIndex++ ] = array[ rightIndex++ ];
96     } /* end if */
97     else { /* if at end of right subarray... */
98        while ( leftIndex <= middle1 ) /* copy the left subarray */
99           tempArray[ combinedIndex++ ] = array[ leftIndex++ ];
100    } /* end else */
101
102    /* copy values back into original array */
103    for ( i = left; i <= right; i++ )
104       array[ i ] = tempArray[ i ];
105
106    /* output merged subarray */
107    printf( "%s* ", "          " );
108    displaySubArray( array, left, right );
109    puts( "\n" );
110 } /* end function merge */
111
112 /* display elements in array */
113 void displayElements( int array[], int length )
114 {
115    displaySubArray( array, 0, length - 1 );
116 } /* end function displayElements */
117
118 /* display certain elements in array */
119 void displaySubArray( int array[], int left, int right )
120 {
121    int i; /* int used in for loop */
122
123    /* output spaces for alignment */
124    for ( i = 0; i < left; i++ )
125       printf( "%s* ", "   " );
126
127    /* output elements left in array */
128    for ( i = left; i <= right; i++ )
129       printf( " %d", array[ i ] );
130 } /* end function displaySubArray */
```

Fig. E.3 | Merge sort algorithm. (Part 3 of 5.)

```
Unsorted array:
 79 86 60 79 76 71 44 88 58 23

split:    79 86 60 79 76 71 44 88 58 23
          79 86 60 79 76
                         71 44 88 58 23

split:    79 86 60 79 76
          79 86 60
                   79 76

split:    79 86 60
          79 86
                60

split:    79 86
          79
             86

merge:    79
             86
          79 86

merge:    79 86
                60
          60 79 86

split:              79 76
                    79
                       76

merge:              79
                       76
                    76 79

merge:    60 79 86
                    76 79
          60 76 79 79 86

split:                     71 44 88 58 23
                           71 44 88
                                    58 23

split:                     71 44 88
                           71 44
                                 88

split:                     71 44
                           71
                              44
merge:                     71
                              44
                           44 71
```

Fig. E.3 | Merge sort algorithm. (Part 4 of 5.)

```
merge:                    44 71
                                88
                          44 71 88

split:                              58 23
                                    58
                                       23

merge:                              58
                                       23
                                    23 58

merge:                    44 71 88
                                    23 58
                          23 44 58 71 88

merge:    60 76 79 79 86
                          23 44 58 71 88
          23 44 58 60 71 76 79 79 86 88

Sorted array:
 23 44 58 60 71 76 79 79 86 88
```

Fig. E.3 | Merge sort algorithm. (Part 5 of 5.)

Lines 85–92 in function `merge` loop until the program reaches the end of either subarray. Line 88 tests which element at the beginning of the arrays is smaller. If the element in the left array is smaller, line 89 places it in position in the combined array. If the element in the right array is smaller, line 91 places it in position in the combined array. When the `while` loop completes, one entire subarray is placed in the combined array, but the other subarray still contains data. Line 94 tests whether the left array has reached the end. If so, lines 95–96 fill the combined array with the elements of the right array. If the left array has not reached the end, then the right array must have reached the end, and lines 99–100 fill the combined array with the elements of the left array. Finally, lines 104–105 copy the combined array into the original array. The output from this program displays the splits and merges performed by merge sort, showing the progress of the sort at each step of the algorithm.

Efficiency of Merge Sort

Merge sort is a far more efficient algorithm than either insertion sort or selection sort (although that may be difficult to believe when looking at the rather busy Fig. E.3). Consider the first (nonrecursive) call to function `sortSubArray`. This results in two recursive calls to function `sortSubArray` with subarrays each approximately half the size of the original array, and a single call to function `merge`. This call to function `merge` requires, at worst, $n - 1$ comparisons to fill the original array, which is $O(n)$. (Recall that each element in the array is chosen by comparing one element from each of the subarrays.) The two calls to function `sortSubArray` result in four more recursive calls to function `sortSubArray`, each with a subarray approximately one quarter the size of the original array, along with two calls to function `merge`. These two calls to the function `merge` each require, at worst, $n/2 - 1$ comparisons, for a total number of comparisons of $O(n)$. This process continues, each call

to `sortSubArray` generating two additional calls to `sortSubArray` and a call to `merge`, until the algorithm has split the array into one-element subarrays. At each level, $O(n)$ comparisons are required to merge the subarrays. Each level splits the size of the arrays in half, so doubling the size of the array requires one more level. Quadrupling the size of the array requires two more levels. This pattern is logarithmic and results in $\log_2 n$ levels. This results in a total efficiency of $O(n \log n)$.

Figure E.4 summarizes many of the searching and sorting algorithms covered in this book and lists the Big O for each of them. Figure E.5 lists the Big O values we've covered in this appendix along with a number of values for n to highlight the differences in the growth rates.

Algorithm	Big O
Insertion sort	$O(n^2)$
Selection sort	$O(n^2)$
Merge sort	$O(n \log n)$
Bubble sort	$O(n^2)$
Quicksort	Worst case: $O(n^2)$ Average case: $O(n \log n)$

Fig. E.4 | Searching and sorting algorithms with Big O values.

n	Approximate decimal value	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	$10 \cdot 2^{10}$	2^{20}
2^{20}	1,000,000	20	2^{20}	$20 \cdot 2^{20}$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$30 \cdot 2^{30}$	2^{60}

Fig. E.5 | Approximate number of comparisons for common Big O notations.

Summary

Section E.1 Introduction

- Sorting involves arranging data into order.

Section E.2 Big O Notation

- One way to describe the efficiency of an algorithm is with Big O notation (O), which indicates how hard an algorithm may have to work to solve a problem.
- For searching and sorting algorithms, Big O describes how the amount of effort of a particular algorithm varies, depending on how many elements are in the data.
- An algorithm that is $O(1)$ is said to have a constant run time. This does not mean that the algorithm requires only one comparison. It just means that the number of comparisons does not grow as the size of the array increases.

- An $O(n)$ algorithm is referred to as having a linear run time.
- Big O is designed to highlight dominant factors and ignore terms that become unimportant with high values of n.
- Big O notation is concerned with the growth rate of algorithm run times, so constants are ignored.

Section E.3 Selection Sort

- Selection sort is a simple, but inefficient, sorting algorithm.
- The first iteration of selection sort selects the smallest element in the array and swaps it with the first element. The second iteration of selection sort selects the second-smallest element (which is the smallest of those remaining) and swaps it with the second element. Selection sort continues until the last iteration selects the second-largest element and swaps it with the second-to-last, leaving the largest element as the last. At the ith iteration of selection sort, the smallest i elements of the whole array are sorted into the first i positions of the array.
- The selection sort algorithm runs in $O(n^2)$ time.

Section E.4 Insertion Sort

- The first iteration of insertion sort takes the second element in the array and, if it's less than the first element, swaps it with the first element. The second iteration of insertion sort looks at the third element and inserts it in the correct position with respect to the first two elements. After the ith iteration of insertion sort, the first i elements in the original array are sorted. Only $n - 1$ iterations are required.
- The insertion sort algorithm runs in $O(n^2)$ time.

Section E.5 Merge Sort

- Merge sort is a sorting algorithm that is faster, but more complex to implement, than selection sort and insertion sort.
- The merge sort algorithm sorts a array by splitting the array into two equal-sized subarrays, sorting each subarray and merging the subarrays into one larger array.
- Merge sort's base case is an array with one element, which is already sorted, so merge sort immediately returns when it's called with a one-element array. The merge part of merge sort takes two sorted arrays (these could be one-element arrays) and combines them into one larger sorted array.
- Merge sort performs the merge by looking at the first element in each array, which is also the smallest element. Merge sort takes the smallest of these and places it in the first element of the larger, sorted array. If there are still elements in the subarray, merge sort looks at the second element in that subarray (which is now the smallest element remaining) and compares it to the first element in the other subarray. Merge sort continues this process until the larger array is filled.
- In the worst case, the first call to merge sort has to make $O(n)$ comparisons to fill the n slots in the final array.
- The merging portion of the merge sort algorithm is performed on two subarrays, each of approximately size $n/2$. Creating each of these subarrays requires $n/2–1$ comparisons for each subarray, or $O(n)$ comparisons total. This pattern continues, as each level works on twice as many arrays, but each is half the size of the previous array.
- This halving results in log n levels, each level requiring $O(n)$ comparisons, for a total efficiency of $O(n \log n)$, which is far more efficient than $O(n^2)$.

Terminology

Big O notation 2
constant run time 2
insertion sort algorithm 7
linear run time 2

merge sort algorithm 10
$O(1)$ 2
$O(n \log n)$ time 16
$O(n)$ time 2
$O(n^2)$ time 3, 6
quadratic run time 3
selection sort algorithm 3

Self-Review Exercises

E.1 Fill in the blanks in each of the following statements:
a) A selection sort application would take approximately _____ times as long to run on a 128-element array as on a 32-element array.
b) The efficiency of merge sort is _____.

E.2 The Big O of the linear search is $O(n)$ and of the binary search is $O(\log n)$. What key aspect of both the binary search (Chapter 6) and the merge sort accounts for the logarithmic portion of their respective Big Os?

E.3 In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?

E.4 In the text, we say that after the merge sort splits the array into two subarrays, it then sorts these two subarrays and merges them. Why might someone be puzzled by our statement that "it then sorts these two subarrays"?

Answers to Self-Review Exercises

E.1 a) 16, because an $O(n^2)$ algorithm takes 16 times as long to sort four times as much information. b) $O(n \log n)$.

E.2 Both of these algorithms incorporate "halving"—somehow reducing something by half on each pass. The binary search eliminates from consideration one-half of the array after each comparison. The merge sort splits the array in half each time it's called.

E.3 The insertion sort is easier to understand and to implement than the merge sort. The merge sort is far more efficient—$O(n \log n)$—than the insertion sort—$O(n^2)$.

E.4 In a sense, it does not really sort these two subarrays. It simply keeps splitting the original array in half until it provides a one-element subarray, which is, of course, sorted. It then builds up the original two subarrays by merging these one-element arrays to form larger subarrays, which are then merged, and so on.

Exercises

E.5 (*Recursive Selection Sort*) A selection sort searches an array looking for the smallest element in the array. When the smallest element is found, it's swapped with the first element of the array. The process is then repeated for the subarray, beginning with the second element of the array. Each pass of the array results in one element being placed in its proper location. This sort requires processing capabilities similar to those of the bubble sort—for an array of n elements, $n - 1$ passes must be made, and for each subarray, $n - 1$ comparisons must be made to find the smallest value. When the subarray being processed contains one element, the array is sorted. Write a recursive function `selectionSort` to perform this algorithm.

E.6 (*Bucket Sort*) A bucket sort begins with a single-subscripted array of positive integers to be sorted, and a double-subscripted array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to $n - 1$, where n is the number of values in the array to be sorted. Each row of the double-subscripted array is referred to as a bucket. Write a function `bucketSort` that takes an integer array and the array size as arguments.

The algorithm is as follows:

a) Loop through the single-subscripted array and place each of its values in a row of the bucket array based on its ones digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0.
b) Loop through the bucket array and copy the values back to the original array. The new order of the above values in the single-subscripted array is 100, 3 and 97.
c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, and so on) and stop when the leftmost digit of the largest number has been processed.

On the second pass of the array, 100 is placed in row 0, 3 is placed in row 0 (it had only one digit so we treat it as 03) and 97 is placed in row 9. The order of the values in the single-subscripted array is 100, 3 and 97. On the third pass, 100 is placed in row 1, 3 (003) is placed in row zero and 97 (097) is placed in row zero (after 3). The bucket sort is guaranteed to have all the values properly sorted after processing the leftmost digit of the largest number. The bucket sort knows it's done when all the values are copied into row zero of the double-subscripted array.

The double-subscripted array of buckets is ten times the size of the integer array being sorted. This sorting technique provides far better performance than a bubble sort but requires much larger storage capacity. Bubble sort requires only one additional memory location for the type of data being sorted. Bucket sort is an example of a space-time trade-off. It uses more memory but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second double-subscripted bucket array and repeatedly move the data between the two bucket arrays until all the data is copied into row zero of one of the arrays. Row zero then contains the sorted array.

E.7 (***Quicksort***) In the examples and exercises of Chapter 6, we discussed the sorting techniques bubble sort, bucket sort and selection sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

a) *Partitioning Step:* Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.
b) *Recursive Step:* Perform *Step a* on each unsorted subarray.

Each time *Step a* is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, it must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subarray? As an example, consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

a) Starting from the rightmost element of the array, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 12, so **37** and 12 are swapped. The new array is

12 2 6 4 89 8 10 **37** 68 45

Element 12 is in italic to indicate that it was just swapped with **37**.

b) Starting from the left of the array, but beginning with the element after 12, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. The first element greater than **37** is 89, so **37** and 89 are swapped. The new array is

12 2 6 4 **37** 8 10 *89* 68 45

c) Starting from the right, but beginning with the element before 89, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The new array is

```
12   2   6   4   10   8   37   89   68   45
```

d) Starting from the left, but beginning with the element after 10, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. There are no more elements greater than **37**, so when we compare **37** with itself, we know that **37** has been placed in its final location in the sorted array.

Once the partition has been applied to the array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues by partitioning both subarrays in the same manner as the original array.

Write recursive function `quicksort` to sort a single-subscripted integer array. The function should receive as arguments an integer array, a starting subscript and an ending subscript. Function `partition` should be called by `quicksort` to perform the partitioning step.