16

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 chapter you will 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 chapter exercises) additional recursive sorts, including quicksort and a recursive version of selection sort.
- To explore (in the chapter exercises) the bucket sort, which achieves very high performance, but by using considerably more memory than the other sorts we have studied—an example of the so-called “space–time trade-off.”

Outline

16.1 Introduction

- **Sorting data**
  - **Place data in order**
    - **Typically ascending or descending**
    - **Based on one or more sort keys**
  - **Algorithms**
    - **Insertion sort**
    - **Selection sort**
    - **Merge sort**
      - **More efficient, but more complex**

16.1 Introduction (Cont.)

- **Big O notation**
  - **Estimates worst-case runtime for an algorithm**
    - **How hard an algorithm must work to solve a problem**

16.2 Big O Notation

- **Big O notation**
  - **Measures runtime growth of an algorithm relative to number of items processed**
    - **Highlights dominant terms**
    - **Ignores terms that become unimportant as *n* grows**
    - **Ignores constant factors**

16.2 Big O Notation (Cont.)

- **Constant runtime**
  - **Number of operations performed by algorithm is constant**
    - **Does not grow as number of items increases**
  - **Represented in Big O notation as *O(1)***
    - **Pronounced “on the order of 1” or “order 1”**
  - **Example**
    - **Test if the first element of an *n*-array is equal to the second element**
      - **Always takes one comparison, no matter how large the array**

16.2 Big O Notation (Cont.)

- **Linear runtime**
  - **Number of operations performed by algorithm grows linearly with number of items**
  - **Represented in Big O notation as *O(n)***
    - **Pronounced “on the order of *n*” or “order *n*”**
  - **Example**
    - **Test if the first element of an *n*-array is equal to any other element**
      - **Takes *n* - 1 comparisons**
      - ***n* term dominates, -1 is ignored**

16.2 Big O Notation (Cont.)

- **Quadratic runtime**
  - **Number of operations performed by algorithm grows as the square of the number of items**
  - **Represented in Big O notation as $O(n^2)$**
    - **Pronounced "on the order of n^2" or "order n^2"**
  - **Example**
    - **Test if any element of an n-array is equal to any other element**
      - **Takes $n^2/2 - n/2$ comparisons**
      - **n^2 term dominates, constant 1/2 is ignored, $-n/2$ is ignored**

16.3 Selection Sort

- **Selection sort**
  - **At *i*th iteration**
    - **Swaps the *i*th smallest element with element *i***
  - **After *i*th iteration**
    - **Smallest *i* elements are sorted in increasing order in first *i* positions**
  - **Requires a total of $(n^2 - n)/2$ comparisons**
    - **Iterates *n* - 1 times**
      - **In *i*th iteration, locating *i*th smallest element requires *n* – *i* comparisons**
    - **Has Big O of $O(n^2)$**

Outline

fig16_01.c

(1 of 4)

```
1  /* Fig. 16.1: fig16_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    printf( "Unsorted array:\n" );
24
25    for ( i = 0; i < SIZE; i++ ) /* print the array */
26       printf( "%d  ", array[ i ] );
27
28    printf( "\n\n" );
29    selectionSort( array, SIZE );
30    printf( "Sorted array:\n" );
```

Outline

fig16_01.c

```
31
32    for ( i = 0; i < SIZE; i++ ) /* print the array */
33       printf( "%d  ", array[ i ] );
34
35    return 0;
36 } /* end function main */
37
38 /* function that selection sorts the array */
39 void selectionSort( int array[], int length )
40 {
41    int smallest; /* index of smallest element */
42    int i, j; /* ints used in for loops */
43
44    /* loop over length - 1 elements */
45    for ( i = 0; i < length - 1; i++ ) {
46       smallest = i; /* first index of remaining array */
47
48       /* loop to find index of smallest element */
49       for ( j = i + 1; j < length; j++ )
50          if ( array[ j ] < array[ smallest ] )
51             smallest = j;
52
53       swap( array, i, smallest ); /* swap smallest element */
54       printPass( array, length, i + 1, smallest ); /* output pass */
55    } /* end for */
56 } /* end function selectionSort */
```

Store the index of the smallest element in the remaining array

Iterate through the whole array length – 1 times

Initializes the index of the smallest element to the current item

Determine the index of the remaining smallest element

Place the smallest remaining element in the next spot

Outline

fig16_01.c

(3 of 4)

```
57
58 /* function that swaps two elements in the array */
59 void swap( int array[], int first, int second )
60 {
61    int temp; /* temporary integer */
62    temp = array[ first ];
63    array[ first ] = array[ second ];
64    array[ second ] = temp;
65 } /* end function swap */
66
67 /* function that prints a pass of the algorithm */
68 void printPass( int array[], int length, int pass, int index )
69 {
70    int i; /* int used in for loop */
71
72    printf( "After pass %2d: ", pass );
73
74    /* output elements till selected item */
75    for ( i = 0; i < index; i++ )
76       printf( "%d  ", array[ i ] );
77
78    printf( "%d* ", array[ index ] ); /* indicate swap */
79
80    /* finish outputting array */
81    for ( i = index + 1; i < length; i++ )
82       printf( "%d  ", array[ i ] );
83
84    printf( "\n               " ); /* for alignment */
85
```

Swap two elements

```
86    /* indicate amount of array that is sorted */
87    for ( i = 0; i < pass; i++ )
88       printf( "--  " );
89
90    printf( "\n" ); /* add newline */
91 } /* 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
```

Outline

fig16_01.c

(4 of 4)

16.4 Insertion Sort

- **Insertion sort**
  - **At *i*th iteration**
    - **Insert (*i* + 1)th element into correct position with respect to first *i* elements**
  - **After *i*th iteration**
    - **First *i* elements are sorted**
  - **Requires a worst-case of n^2 inner-loop iterations**
    - **Outer loop iterates *n* - 1 times**
      - **Inner loop requires *n* – 1iterations in worst case**
    - **For determining Big O, nested statements mean multiply the number of iterations**
    - **Has Big O of $O(n^2)$**

Outline

fig16_02.c

(1 of 4)

```
1  /* Fig. 16.2: fig16_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    printf( "Unsorted array:\n" );
23
```

Outline

fig16_02.c

(2 of 4)

```
24    for ( i = 0; i < SIZE; i++ ) /* print the array */
25       printf( "%d  ", array[ i ] );
26
27    printf( "\n\n" );
28    insertionSort( array, SIZE );
29    printf( "Sorted array:\n" );
30
31    for ( i = 0; i < SIZE; i++ ) /* print the array */
32       printf( "%d  ", array[ i ] );
33
34    return 0;
35 } /* end function main */
36
37 /* function that sorts the array */
38 void insertionSort( int array[], int length )
39 {
40    int insert; /* temporary variable to hold element to insert */
41    int i; /* int used in for loop */
42
43    /* loop over length - 1 elements */
44    for ( i = 1; i < length; i++ ) {
45       int moveItem = i; /* initialize location to place element */
46       insert = array[ i ];
47
48       /* search for place to put current element */
49       while ( moveItem > 0 && array[ moveItem - 1 ] > insert ) {
50          /* shift element right one slot */
51          array[ moveItem ] = array[ moveItem - 1 ];
52          --moveItem;
53       } /* end while */
```

Holds the element to be inserted while the order elements are moved

Iterate through length - 1 items in the array

Keep track of where to insert the element

Stores the value of the element that will be inserted in the sorted portion of the array

Loop to locate the correct position to insert the element

Moves an element to the right and decrement the position at which to insert the next element

```
54
55          array[ moveItem ] = insert; /* place inserted element */
56          printPass( array, length, i, moveItem );
57       } /* end for */
58    } /* end function insertionSort */
59
60    /* function that prints a pass of the algorithm */
61    void printPass( int array[], int length, int pass, int index )
62    {
63       int i; /* int used in for loop */
64
65       printf( "After pass %2d: ", pass );
66
67       /* output elements till selected item */
68       for ( i = 0; i < index; i++ )
69          printf( "%d  ", array[ i ] );
70
71       printf( "%d* ", array[ index ] ); /* indicate swap */
72
73       /* finish outputting array */
74       for ( i = index + 1; i < length; i++ )
75          printf( "%d  ", array[ i ] );
76
77       printf( "\n               " ); /* for alignment */
78
79       /* indicate amount of array that is sorted */
80       for ( i = 0; i <= pass; i++ )
81          printf( "--  " );
```

Inserts the element in place

Outline

fig16_02.c

(3 of 4)

```
82
83    printf( "\n" ); /* add newline */
84 } /* 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
               --  --  --  --  --  --
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
```

Outline

fig16_02.c

(4 of 4)

16.5 Merge Sort

- **Merge sort**
  - **Sorts array by**
    - **Splitting it into two equal-size subarrays**
      - **If array size is odd, one subarray will be one element larger than the other**
    - **Sorting each subarray**
    - **Merging them into one larger, sorted array**
      - **Repeatedly compare smallest elements in the two subarrays**
      - **The smaller element is removed and placed into the larger, combined array**

16.5 Merge Sort (Cont.)

- **Our recursive implementation**
  - **Base case**
    - **An array with one element is already sorted**
  - **Recursion step**
    - **Split the array (of ≥ 2 elements) into two equal halves**
      - **If array size is odd, one subarray will be one element larger than the other**
    - **Recursively sort each subarray**
    - **Merge them into one larger, sorted array**

16.5 Merge Sort (Cont.)

- **Merge sort (Cont.)**
  - **Sample merging step**
    - **Smaller, sorted arrays**
      - **A: `4 10 34 56 77`**
      - **B: `5 30 51 52 93`**
    - **Compare smallest element in A to smallest element in B**
      - **`4` (A) is less than `5` (B)**
        - **`4` becomes first element in merged array**
      - **`5` (B) is less than `10` (A)**
        - **`5` becomes second element in merged array**
      - **`10` (A) is less than `30` (B)**
        - **`10` becomes third element in merged array**
      - **Etc.**

Outline

fig16_03.c

(1 of 8)

```
1  /* Fig. 16.3: fig16_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
```

Outline

fig16_03.c

(2 of 8)

```
25    printf( "Unsorted array:\n" );
26    displayElements( array, SIZE ); /* print the array */
27    printf( "\n\n" );
28    mergeSort( array, SIZE ); /* merge sort the array */
29    printf( "Sorted array:\n" );
30    displayElements( array, SIZE ); /* print the array */
31    return 0;
32 } /* end function main */
33
34 /* function that merge sorts the array */
35 void mergeSort( int array[], int length )
36 {
37    sortSubArray( array, 0, length - 1 );
38 } /* end function mergeSort */
39
40 /* function that sorts a piece of the array */
41 void sortSubArray( int array[], int low, int high )
42 {
43    int middle1, middle2; /* ints that record where the array is split */
44
45    /* test base case: size of array is 1 */
46    if ( ( high - low ) >= 1 ) { /* if not base case... */
47       middle1 = ( low + high ) / 2;
48       middle2 = middle1 + 1;
49
```

Call function **sortSubArray** with **0** and **length – 1** as the beginning and ending indices

Test the base case

Split the array in two

Outline

fig16_03.c

```
50       /* output split step */
51       printf( "split:   " );
52       displaySubArray( array, low, high );
53       printf( "\n         " );
54       displaySubArray( array, low, middle1 );
55       printf( "\n         " );
56       displaySubArray( array, middle2, high );
57       printf( "\n\n" );
58
59       /* split array in half and sort each half recursively */
60       sortSubArray( array, low, middle1 ); /* first half */
61       sortSubArray( array, middle2, high ); /* second half */
62
63       /* merge the two sorted arrays */
64       merge( array, low, middle1, middle2, high );
65    } /* end if */
66 } /* end function sortSubArray */
67
68 /* merge two sorted subarrays into one sorted subarray */
69 void merge( int array[], int left, int middle1, int middle2, int right )
70 {
71    int leftIndex = left; /* index into left subarray */
72    int rightIndex = middle2; /* index into right subarray */
73    int combinedIndex = left; /* index into temporary array */
74    int tempArray[ SIZE ]; /* temporary array */
75    int i; /* int used in for loop */
76
```

Recursively call function **sortSubArray** on the two subarrays

Combine the two sorted arrays into one larger, sorted array

Outline

```
77    /* output two subarrays before merging */
78    printf( "merge:    " );
79    displaySubArray( array, left, middle1 );
80    printf( "\n          " );
81    displaySubArray( array, middle2, right );
82    printf( "\n" );
83
84    /* merge the subarrays until the end of one is reached */
85    while ( leftIndex <= middle1 && rightIndex <= right ) {
86       /* place the smaller of the two current elements in result */
87       /* and move to the next space in the subarray */
88       if ( array[ leftIndex ] <= array[ rightIndex ] )
89          tempArray[ combinedIndex++ ] = array[ leftIndex++ ];
90       else
91          tempArray[ combinedIndex++ ] = array[ rightIndex++ ];
92    } /* end while */
93
94    if ( leftIndex == middle2 ) { /* if at end of left subarray ... */
95       while ( rightIndex <= right ) /* copy the right subarray */
96          tempArray[ combinedIndex++ ] = array[ rightIndex++ ];
97    } /* end if */
98    else { /* if at end of right subarray... */
99       while ( leftIndex <= middle1 ) /* copy the left subarray */
100         tempArray[ combinedIndex++ ] = array[ leftIndex++ ];
101   } /* end else */
102
103   /* copy values back into original array */
104   for ( i = left; i <= right; i++ )
105      array[ i ] = tempArray[ i ];
106
```

Loop until the end of either subarray is reached

fig16_03.c

Test which element at the beginning of the arrays is smaller

Place the smaller element in the combined array

Fill the combined array with the remaining elements of the right array or…

…else fill the combined array with the remaining elements of the left array

Copy the combined array into the original array

Outline

fig16_03.c

(5 of 8)

```
107   /* output merged subarray */
108   printf( "         " );
109   displaySubArray( array, left, right );
110   printf( "\n\n" );
111 } /* end function merge */
112
113 /* display elements in array */
114 void displayElements( int array[], int length )
115 {
116   displaySubArray( array, 0, length - 1 );
117 } /* end function displayElements */
118
119 /* display certain elements in array */
120 void displaySubArray( int array[], int left, int right )
121 {
122   int i; /* int used in for loop */
123
124   /* output spaces for alignment */
125   for ( i = 0; i < left; i++ )
126      printf( "   " );
127
128   /* output elements left in array */
129   for ( i = left; i <= right; i++ )
130      printf( " %d", array[ i ] );
131 } /* end function displaySubArray */
```

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

(continued on next slide...)

(continued from previous slide…)

```
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
```

(continued on next slide...)

Outline

fig16_03.c

(6 of 8)

(continued from previous slide…)

```
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
merge:                    44 71
                                88
                          44 71 88

split:                             58 23
                                   58
                                      23
```

(continued on next slide...)

Outline

fig16_03.c

(7 of 8)

(continued from previous slide…)

```
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
```

Outline

fig16_03.c

(8 of 8)

16.5 Merge Sort (Cont.)

- **Efficiency of merge sort**
  - ***n* log *n* runtime**
    - **arrays means log $_2$ *n* levels to reach base case**
      - **Doubling size of array requires one more level**
      - **Quadrupling size of array requires two more levels**
    - ***O(n)* comparisons are required at each level**
      - **Calling `sortSubArray` with a size-*n* array results in**
        - **Two `sortSubArray` calls with size-*n*/2 subarrays**
        - **A `merge` operation with *n* – 1 (order *n*) comparisons**
      - **So, always order *n* total comparisons at each level**
    - **Represented in Big O notation as *O(n log n)***
      - **Pronounced "on the order of *n* log *n*" or "order *n* log *n*"**

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. 16.4 | 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 . 2^{10}	2^{20}
2^{20}	1,000,000	20	2^{20}	20 . 2^{20}	2^{40}
2^{30}	1,000,000,000	30	2^{30}	30 . 2^{30}	2^{60}

Fig. 16.5 | Approximate number of comparisons for common Big O notations.