



25

# Searching and Sorting



# OBJECTIVES

In this chapter you will learn:

- To search for a given value in an array using the linear search and binary search algorithm.
- To sort arrays using the iterative selection and insertion sort algorithms.
- To sort arrays using the recursive merge sort algorithm.
- To determine the efficiency of searching and sorting algorithms.



## 25.1 Introduction

- **Searching** data involves determining whether a **search key** is present in the data and finding its location.
  - Two popular search algorithms are the simple linear search and the binary search.
- **Sorting** places data in order, based on one or more **sort keys**.



## 25.2 Searching Algorithms

### 25.2.1 Linear Search

- Consider an array containing the following values and an application that is searching for 51:

34    56    2    10    77    51    93    30    5    52

- Using the **linear search algorithm**, the application checks the array sequentially and returns the index 5.
- If the application determines that the search key does not match any element in the array, the application returns a sentinel value (e.g. -1).



## Outline

### LinearArray.cs

(1 of 3)

```
1 // Fig. 25.2: LinearArray.cs
2 // Class that contains an array of random integers and a method
3 // that will search that array sequentially.
4 using System;
5
6 public class LinearArray
7 {
8     private int[] data; // array of values
9     private static Random generator = new Random();
10
11     // create array of given size and fill with random integers
12     public LinearArray( int size )
13     {
14         data = new int[ size ]; // create space for array
15     }
```

**Fig. 25.2** | Class that contains an array of random integers and a method that searches that array sequentially. (Part 1 of 3).



## Outline

### LinearArray.cs

(2 of 3)

```
16     // fill array with random ints in range 10-99
17     for ( int i = 0; i < size; i++ )
18         data[ i ] = generator.Next( 10, 100 );
19 } // end LinearArray constructor
20
21 // perform a linear search on the data
22 public int LinearSearch( int searchKey )
23 {
24     // loop through array sequentially
25     for ( int index = 0; index < data.Length; index++ )
26         if ( data[ index ] == searchKey )
27             return index; // return index of integer
28
29     return -1; // integer was not found
30 } // end method LinearSearch
31
```

**Fig. 25.2** | Class that contains an array of random integers and a method that searches that array sequentially. (Part 2 of 3).



## Outline

### LinearArray.cs

(3 of 3)

```
32 // method to output values in array
33 public override string ToString()
34 {
35     string temporary = string.Empty;
36
37     // iterate through array
38     foreach ( int element in data )
39         temporary += element + " ";
40
41     temporary += "\n"; // add newline character
42     return temporary;
43 } // end method ToString
44 } // end class LinearArray
```

**Fig. 25.2** | Class that contains an array of random integers and a method that searches that array sequentially. (Part 3 of 3).



## 25.2 Searching Algorithms (Cont.)

- The major difference between search algorithms is the amount of effort required to complete the search.
- **Big O notation** is a measure of the worst-case runtime for an algorithm.



## 25.2 Searching Algorithms (Cont.)

- An algorithm tests whether the first element of an array is equal to the second element.
  - If the array has 10 or 1,000 elements, this algorithm requires one comparison.
- This algorithm is said to have a **constant runtime**, which is represented as  $O(1)$ .
- $O(1)$  does not necessarily require only one comparison.  $O(1)$  just means that the number of comparisons is *constant*.



## 25.2 Searching Algorithms (Cont.)

- An algorithm tests whether the first element of an array is equal to *any* of the other elements.
  - If the array has 10 elements, this algorithm requires up to nine comparisons.
  - If the array has 1,000 elements, this algorithm requires up to 999 comparisons.



## 25.2 Searching Algorithms (Cont.)

- As the number of elements being sorted ( $n$ ) grows larger, the  $n$  part of the expression “dominates.”
- 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 runtime**.



## 25.2 Searching Algorithms (Cont.)

- Suppose an algorithm tests whether *any* element of an array is duplicated in the array.
  - The first element must be compared with every other element.
  - The second element must be compared with every element except the first.
- This algorithm will make  $(n - 1) + (n - 2) + \dots + 2 + 1$  or  $n^2/2 - n/2$  comparisons.
- As  $n$  increases, the  $n$  term becomes inconsequential.



## 25.2 Searching Algorithms (Cont.)

- In an algorithm requiring  $n^2$  comparisons, doubling the number of elements quadruples the number of comparisons.
- These algorithms grow as the square of  $n$ , so they are considered to be  $O(n^2)$ , referred to as having **quadratic runtime**.
- Algorithms with more favorable Big O measures take cleverness and effort to create, but their superior performance can be well worth the effort.

### Performance Tip 25.1

**Sometimes the simplest algorithms perform poorly. Their virtue is that they are easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.**



## 25.2 Searching Algorithms (Cont.)

### 25.2.2 Binary Search

- The **binary search algorithm** is more efficient than the linear search algorithm, but it requires that the array be sorted.
- The first iteration tests the middle element in the array.
- If the search key is less than the middle element, the algorithm continues with only the first half of the array.
- Each iteration tests the middle value of the remaining portion of the array, called a **subarray**.



## 25.2 Searching Algorithms (Cont.)

- Consider this sorted 15-element array and a search key of 65:

2 3 5 10 27 30 34 51 56 65 77 81 82 93 99

- Since search key (65) is larger than the middle element (51), the first half of the array is discarded.
- The algorithm then checks whether 81 (the middle element of the subarray) matches the search key.
- The algorithm then checks 65 (which indeed matches the search key).



## Outline

- Class `BinaryArray` (Fig. 25.4) is similar to `LinearArray`.
- Method `Sort` is a `static` method of class `Array` that sorts the elements in an array in ascending order.

`BinaryArray.cs`

(1 of 4)

```
1 // Fig. 25.4: BinaryArray.cs
2 // Class that contains an array of random integers and a method
3 // that uses binary search to find an integer.
4 using System;
5
6 public class BinaryArray
7 {
8     private int[] data; // array of values
9     private static Random generator = new Random();
10
11     // create array of given size and fill with random integers
12     public BinaryArray( int size )
13     {
14         data = new int[ size ]; // create space for array
15
```

**Fig. 25.4** | Class that contains an array of random integers and a method that uses binary search to find an integer. (Part 1 of 4).



## Outline

### BinaryArray.cs

(2 of 4)

```

16 // fill array with random ints in range 10-99
17 for ( int i = 0; i < size; i++ )
18     data[ i ] = generator.Next( 10, 100 );
19
20     Array.Sort( data );
21 } // end BinaryArray constructor
22
23 // perform a binary search on the data
24 public int BinarySearch( int searchElement )
25 {
26     int low = 0; // low end of the search area
27     int high = data.Length - 1; // high end of the search area
28     int middle = ( low + high + 1 ) / 2; // middle element
29     int location = -1; // return value; -1 if not found
30
31     do // loop to search for element
32     {
33         // print remaining elements of array
34         Console.Write( RemainingElements( low, high ) );
35

```

Initializing the array with random ints.

Using `Array.Sort` to order the data array.

Calculating the low, high and middle indices of the current search array.

`location` is set to the sentinel value `-1`.

**Fig. 25.4** | Class that contains an array of random integers and a method that uses binary search to find an integer. (Part 2 of 4).



## Outline

### BinaryArray.cs

(3 of 4)

```

36     // output spaces for alignment
37     for ( int i = 0; i < middle; i++ )
38         console.write( "  " );
39
40     Console.WriteLine( " * " ); // indicate current middle
41
42     // if the element is found at the middle
43     if ( searchElement == data[ middle ] )
44         location = middle; // location is the current middle
45
46     // middle element is too high
47     else if ( searchElement < data[ middle ] )
48         high = middle - 1; // eliminate the higher half
49     else // middle element is too low
50         low = middle + 1; // eliminate the lower half
51
52     middle = ( low + high + 1 ) / 2; // recalculate the middle
53 } while ( ( low <= high ) && ( location == -1 ) );
54
55 return location; // return location of search key
56 } // end method BinarySearch
57

```

Setting location if  
middle is the search key.

Testing for low being greater  
than high (the search key is  
not present) or location does  
not equal -1 (the search key  
was found).

**Fig. 25.4** | Class that contains an array of random integers and a method that uses binary search to find an integer. (Part 3 of 4).



Outline

```

58 // method to output certain values in array
59 public string RemainingElements( int low, int high )
60 {
61     string temporary = string.Empty;
62
63     // output spaces for alignment
64     for ( int i = 0; i < low; i++ )
65         temporary += " ";
66
67     // output elements left in array
68     for ( int i = low; i <= high; i++ )
69         temporary += data[ i ] + " ";
70
71     temporary += "\n";
72     return temporary;
73 } // end method RemainingElements
74
75 // method to output values in array
76 public override string ToString()
77 {
78     return RemainingElements( 0, data.Length - 1 );
79 } // end method ToString
80 } // end class BinaryArray

```

BinaryArray.cs

(4 of 4)

**Fig. 25.4** | Class that contains an array of random integers and a method that uses binary search to find an integer. (Part 4 of 4).



## Outline

- Figure 25.5 tests `BinarySearch` by finding the user's input value in a sequence.

### BinarySearch Test.cs

(1 of 3)

```
1 // Fig. 25.5: BinarySearchTest.cs
2 // Using binary search to locate an item in an array.
3 using System;
4
5 public class BinarySearchTest
6 {
7     public static void Main( string[] args )
8     {
9         int searchInt; // search key
10        int position; // location of search key in array
11
12        // create array and output it
13        BinaryArray searchArray = new BinaryArray( 15 );
14        Console.WriteLine( searchArray );
15
16        // prompt and input first int from user
17        Console.Write( "Please enter an integer value (-1 to quit): " );
18        searchInt = Convert.ToInt32( Console.ReadLine() );
19        Console.WriteLine();
```

**Fig. 25.5** | Using binary search to locate an item in an array. (Part 1 of 3).



## Outline

### BinarySearch Test.cs

(2 of 3)

```
20
21 // repeatedly input an integer; -1 terminates the application
22 while ( searchInt != -1 )
23 {
24     // use binary search to try to find integer
25     position = searchArray.BinarySearch( searchInt );
26
27     // return value of -1 indicates integer was not found
28     if ( position == -1 )
29         Console.WriteLine( "The integer {0} was not found.\n",
30             searchInt );
31     else
32         Console.WriteLine(
33             "The integer {0} was found in position {1}.\n",
34             searchInt, position);
35
36     // prompt and input next int from user
37     Console.Write( "Please enter an integer value (-1 to quit): " );
```

**Fig. 25.5** | Using binary search to locate an item in an array. (Part 2 of 3).



## Outline

### BinarySearch Test.cs

(3 of 3)

```

38         searchInt = Convert.ToInt32( Console.ReadLine() );
39         Console.WriteLine();
40     } // end while
41 } // end Main
42 } // end class BinarySearchTest

```

```

12 17 22 25 30 39 40 52 56 72 76 82 84 91 93
Please enter an integer value (-1 to quit): 72
12 17 22 25 30 39 40 52* 56 72 76 82 84 91 93
                    56 72 76 82* 84 91 93
                        56 72* 76
The integer 72 was found in position 9.
Please enter an integer value (-1 to quit): 13
12 17 22 25 30 39 40 52* 56 72 76 82 84 91 93
12 17 22 25* 30 39 40
12 17* 22
12*
The integer 13 was not found.
Please enter an integer value (-1 to quit): -1

```

**Fig. 25.5** | Using binary search to locate an item in an array. (Part 3 of 3).



## 25.2 Searching Algorithms (Cont.)

- In the worst-case scenario, searching a sorted array of 1,023 elements will take only 10 comparisons when using a binary search.
- The maximum number of comparisons needed is the exponent of the first power of 2 greater than the number of elements.
  - This results in a big O of  $O(\log n)$ , which is also known as **logarithmic runtime**.



## 25.3 Sorting Algorithms

- Sorting data is one of the most important computing applications.
- It is important to understand that the sorted array will be the same no matter which (correct) algorithm you use.
- The choice of algorithm affects only the runtime and memory use of the application.



## 25.3 Sorting Algorithms (Cont.)

### 25.3.1 Selection Sort

- **Selection sort** is a simple, but inefficient, sorting algorithm.
- The first iteration selects the smallest element in the array and swaps it with the first element.
- The second iteration selects the second-smallest element and swaps it with the second element.
- After the  $i$ th iteration, the smallest  $i$  elements of the array will be sorted in increasing order in the first  $i$  positions of the array.



## 25.3 Sorting Algorithms (Cont.)

34    56    4    10    77    51    93    30    5    52

- The selection sort determines the smallest element (4), then swaps it with 34:

4    56    34    10    77    51    93    30    5    52

- The algorithm then determines the smallest value of the remaining elements (5) and swaps it with 56 :

4    5    34    10    77    51    93    30    56    52

- The process continues until the array is fully sorted.

4    5    10    34    77    51    93    30    56    52

4    5    10    30    34    51    52    56    77    93



## Outline

### selectionSort.cs

(1 of 4)

```
1 // Fig. 25.6: SelectionSort.cs
2 // Class that creates an array filled with random integers.
3 // Provides a method to sort the array with selection sort.
4 using System;
5
6 public class SelectionSort
7 {
8     private int[] data; // array of values
9     private static Random generator = new Random();
10
11     // create array of given size and fill with random integers
12     public SelectionSort( int size )
13     {
14         data = new int[ size ]; // create space for array
15
16         // fill array with random ints in range 10-99
17         for ( int i = 0; i < size; i++ )
18             data[ i ] = generator.Next( 10, 100 );
19     } // end SelectionSort constructor
```

The constructor creates and initializes an array with random ints.

**Fig. 25.6** | Class that creates an array filled with random integers. Provides a method to sort the array with selection sort. (Part 1 of 4).



## Outline

### selectionSort.cs

(2 of 4 )

```

20
21 // sort array using selection sort
22 public void Sort()
23 {
24     int smallest; // index of smallest element
25
26     // loop over data.Length - 1 elements
27     for ( int i = 0; i < data.Length - 1; i++ )
28     {
29         smallest = i; // first index of remaining array
30
31         // loop to find index of smallest element
32         for ( int index = i + 1; index < data.Length; index++ )
33             if ( data[ index ] < data[ smallest ] )
34                 smallest = index;
35
36         Swap( i, smallest ); // swap smallest element into position
37         PrintPass( i + 1, smallest ); // output pass of algorithm
38     } // end outer for
39 } // end method Sort
40

```

smallest stores the index of the smallest element in the remaining array.

Looping over the remaining elements and comparing their values to the smallest element.

Calling method Swap to place the smallest remaining element in the next spot in the array.

**Fig. 25.6** | Class that creates an array filled with random integers. Provides a method to sort the array with selection sort. (Part 2 of 4).



## Outline

### selectionSort.cs

(3 of 4)

```
41 // helper method to swap values in two elements
42 public void Swap( int first, int second )
43 {
44     int temporary = data[ first ]; // store first in temporary
45     data[ first ] = data[ second ]; // replace first with second
46     data[ second ] = temporary; // put temporary in second
47 } // end method Swap
48
49 // print a pass of the algorithm
50 public void PrintPass( int pass, int index )
51 {
52     Console.Write( "after pass {0}: ", pass );
53
54     // output elements through the selected item
55     for ( int i = 0; i < index; i++ )
56         Console.Write( data[ i ] + " " );
57
58     Console.Write( data[ index ] + "* " ); // indicate swap
59
60     // finish outputting array
61     for ( int i = index + 1; i < data.Length; i++ )
62         Console.Write( data[ i ] + " " );
```

**Fig. 25.6** | Class that creates an array filled with random integers. Provides a method to sort the array with selection sort. (Part 3 of 4).



## Outline

### selectionSort.cs

(4 of 4)

```
63
64     Console.WriteLine( "\n          " ); // for alignment
65
66     // indicate amount of array that is sorted
67     for( int j = 0; j < pass; j++ )
68         Console.WriteLine( "-- " );
69     Console.WriteLine( "\n" ); // skip a line in output
70 } // end method PrintPass
71
72 // method to output values in array
73 public override string ToString()
74 {
75     string temporary = string.Empty;
76
77     // iterate through array
78     foreach ( int element in data )
79         temporary += element + " ";
80
81     temporary += "\n"; // add newline character
82     return temporary;
83 } // end method ToString
84 } // end class selectionSort
```

**Fig. 25.6** | Class that creates an array filled with random integers. Provides a method to sort the array with selection sort. (Part 4 of 4).



## Outline

- Figure 25.7 demonstrates the operation of a `SelectionSort` object with 10 elements.

### SelectionSort Test.cs

(1 of 2)

```

1 // Fig. 25.7: SelectionSortTest.cs
2 // Testing the selection sort class.
3 using System;
4
5 public class SelectionSortTest
6 {
7     public static void Main( string[] args )
8     {
9         // create object to perform selection sort
10        SelectionSort sortArray = new SelectionSort( 10 );
11
12        Console.WriteLine( "Unsorted array:" );
13        Console.WriteLine( sortArray ); // print unsorted array
14
15        sortArray.Sort(); // sort array
16
17        Console.WriteLine( "Sorted array:" );
18        Console.WriteLine( sortArray ); // print sorted array
19    } // end Main
20 } // end class SelectionSortTest

```

Implicitly calling the object's `ToString` method before sorting.

Calling the `Sort` method to perform the selection sort.

Outputting the object's sorted array.

**Fig. 25.7** | Testing the selection sort class. (Part 1 of 2).



OutlineselectionSort  
Test.cs

(2 of 2)

```

Unsorted array:
86 97 83 45 19 31 86 13 57 61

after pass 1: 13 97 83 45 19 31 86 86* 57 61
              --
after pass 2: 13 19 83 45 97* 31 86 86 57 61
              -- --
after pass 3: 13 19 31 45 97 83* 86 86 57 61
              -- -- --
after pass 4: 13 19 31 45* 97 83 86 86 57 61
              -- -- -- --
after pass 5: 13 19 31 45 57 83 86 86 97* 61
              -- -- -- -- --
after pass 6: 13 19 31 45 57 61 86 86 97 83*
              -- -- -- -- --
after pass 7: 13 19 31 45 57 61 83 86 97 86*
              -- -- -- -- --
after pass 8: 13 19 31 45 57 61 83 86* 97 86
              -- -- -- -- --
after pass 9: 13 19 31 45 57 61 83 86 86 97*
              -- -- -- -- --

Sorted array:
13 19 31 45 57 61 83 86 86 97

```

**Fig. 25.7** | Testing the selection sort class. (Part 2 of 2).



## 25.3 Sorting Algorithms (Cont.)

- The outer for loop iterates over the first  $n - 1$  elements in the array.
- The inner for loop iterates over each element in the remaining array.
- The inner loop iterates a total of  $n(n - 1) / 2$  or  $(n^2 - n)/2$ .
- The selection sort algorithm runs in  $O(n^2)$  time.



## 25.3 Sorting Algorithms (Cont.)

### 25.3.2 Insertion Sort

- **Insertion sort** is a simple, but inefficient, sorting algorithm.
- Its first iteration takes the second element in the array and, if it is less than the first, swaps them.
- The second iteration looks at the third element and determines whether to swap it with a preceding element.
- At the  $i$ th iteration of this algorithm, the first  $i$  elements in the original array will be sorted.



## 25.3 Sorting Algorithms (Cont.)

34    56    4    10    77    51    93    30    5    52

- The algorithm first looks at 34 and 56, and accepts the order.
- In the next iteration, 4 is less than 56, so the application swaps the elements.
- The application then determines that 4 is less than 34, so it moves 34 one element to the right.



## 25.3 Sorting Algorithms (Cont.)

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

- Using this algorithm, at the  $i$ th iteration, the first  $i$  elements of the original array are sorted.
- They may not be in their final locations because smaller values may be located later in the array.



## Outline

- Figure 25.8 declares the InsertionSort class.

```
1 // Fig. 25.8: InsertionSort.cs
2 // Class that creates an array filled with random integers.
3 // Provides a method to sort the array with insertion sort.
4 using System;
5
6 public class InsertionSort
7 {
8     private int[] data; // array of values
9     private static Random generator = new Random();
10
11     // create array of given size and fill with random integers
12     public InsertionSort( int size )
13     {
14         data = new int[ size ]; // create space for array
15
16         // fill array with random ints in range 10-99
17         for ( int i = 0; i < size; i++ )
18             data[ i ] = generator.Next( 10, 100 );
19     } // end InsertionSort constructor
20
```

### InsertionSort.cs

(1 of 4)

**Fig. 25.8** | Class that creates an array filled with random integers.  
Provides a method to sort the array with insertion sort. (Part 1 of 4).



## Outline

```

21 // sort array using insertion sort
22 public void sort()
23 {
24     int insert; // temporary variable to hold element to insert
25
26     // loop over data.Length - 1 elements
27     for ( int next = 1; next < data.Length; next++ )
28     {
29         // store value in current element
30         insert = data[ next ];
31
32         // initialize location to place element
33         int moveItem = next;
34
35         // search for place to put current element
36         while ( moveItem > 0 && data[ moveItem - 1 ] > insert )
37         {
38             // shift element right one slot
39             data[ moveItem ] = data[ moveItem - 1 ];
40             moveItem--;
41         } // end while

```

### InsertionSort.cs

(2 of 4)

Variable `insert` holds the element to be inserted while the other elements are moved.

Temporarily storing the value of the element that will be inserted in the sorted portion of the array.

`moveItem` keeps track of where to insert the element.

Locating the correct position in the sorted section to insert the element.

**Fig. 25.8** | Class that creates an array filled with random integers. Provides a method to sort the array with insertion sort. (Part 2 of 4).



## Outline

### InsertionSort.cs

(3 of 4)

Inserting the element in place.

```
42         data[ moveItem ] = insert; // place inserted element
43         PrintPass( next, moveItem ); // output pass of algorithm
44     } // end for
45 } // end method Sort
46
47 // print a pass of the algorithm
48 public void PrintPass( int pass, int index )
49 {
50     Console.Write( "after pass {0}: ", pass );
51
52     // output elements till swapped item
53     for ( int i = 0; i < index; i++ )
54         Console.Write( data[ i ] + " " );
55
56     Console.Write( data[ index ] + "* " ); // indicate swap
57
58     // finish outputting array
59     for ( int i = index + 1; i < data.Length; i++ )
60         Console.Write( data[ i ] + " " );
61
62
```

**Fig. 25.8** | Class that creates an array filled with random integers. Provides a method to sort the array with insertion sort. (Part 3 of 4).



## Outline

### InsertionSort.cs

(4 of 4)

```
63     Console.WriteLine( "\n          " ); // for alignment
64
65     // indicate amount of array that is sorted
66     for( int i = 0; i <= pass; i++ )
67         Console.WriteLine( "-- " );
68     Console.WriteLine( "\n" ); // skip a line in output
69 } // end method PrintPass
70
71 // method to output values in array
72 public override string ToString()
73 {
74     string temporary = string.Empty;
75
76     // iterate through array
77     foreach ( int element in data )
78         temporary += element + " ";
79
80     temporary += "\n"; // add newline character
81     return temporary;
82 } // end method ToString
83 } // end class InsertionSort
```

**Fig. 25.8** | Class that creates an array filled with random integers.  
Provides a method to sort the array with insertion sort. (Part 4 of 4).



## Outline

- Figure 25.9 is the same as Fig. 25.7 except that it creates and uses an `InsertionSort` object.

### InsertionSort Test.cs

(1 of 2)

```
1 // Fig. 25.9: InsertionSortTest.cs
2 // Testing the insertion sort class.
3 using System;
4
5 public class InsertionSortTest
6 {
7     public static void Main( string[] args )
8     {
9         // create object to perform insertion sort
10        InsertionSort sortArray = new InsertionSort( 10 );
11
12        Console.WriteLine( "Unsorted array:" );
13        Console.WriteLine( sortArray ); // print unsorted array
14
15        sortArray.Sort(); // sort array
16
17        Console.WriteLine( "Sorted array:" );
18        Console.WriteLine( sortArray ); // print sorted array
19    } // end Main
20 } // end class InsertionSortTest
```

**Fig. 25.9** | Testing the insertion sort class. (Part 1 of 2).



Outline

Unsorted array:

12 27 36 28 33 92 11 93 59 62

after pass 1: 12 27\* 36 28 33 92 11 93 59 62

after pass 2: 12 27 36\* 28 33 92 11 93 59 62

after pass 3: 12 27 28\* 36 33 92 11 93 59 62

after pass 4: 12 27 28 33\* 36 92 11 93 59 62

after pass 5: 12 27 28 33 36 92\* 11 93 59 62

after pass 6: 11\* 12 27 28 33 36 92 93 59 62

after pass 7: 11 12 27 28 33 36 92 93\* 59 62

after pass 8: 11 12 27 28 33 36 59\* 92 93 62

after pass 9: 11 12 27 28 33 36 59 62\* 92 93

Sorted array:

11 12 27 28 33 36 59 62 92 93

InsertionSort  
Test.cs

(2 of 2)

**Fig. 25.9** | Testing the insertion sort class. (Part 2 of 2).



## 25.3 Sorting Algorithms (Cont.)

- Like selection sort, the implementation of insertion sort contains nested loops.
- 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.
- The insertion sort algorithm also runs in  $O(n^2)$  time.



## 25.3 Sorting Algorithms (Cont.)

### 25.3.3 Merge Sort

- **Merge sort** is an efficient sorting algorithm but is more complex.
- The merge sort algorithm sorts an array by splitting it into two equal-sized subarrays, sorting each subarray and merging them in one larger array.
- The implementation of merge sort in this example is recursive.



## 25.3 Sorting Algorithms (Cont.)

- Suppose the algorithm has sorted arrays A:

4    10    34    56    77

- and B:

5    30    51    52    93

- The smallest element in A is 4. The smallest element in B is 5.
- The value from A is smaller, so 4 becomes the first element in the merged array.
- The algorithm continues by comparing 10 to 5, and so on.



## Outline

```
1 // Fig. 25.10: MergeSort.cs
2 // Class that creates an array filled with random integers.
3 // Provides a method to sort the array with merge sort.
4 using System;
5
6 public class MergeSort
7 {
8     private int[] data; // array of values
9     private static Random generator = new Random();
10
11     // create array of given size and fill with random integers
12     public MergeSort( int size )
13     {
14         data = new int[ size ]; // create space for array
15
16         // fill array with random ints in range 10-99
17         for ( int i = 0; i < size; i++ )
18             data[ i ] = generator.Next( 10, 100 );
19     } // end MergeSort constructor
20
```

### InsertionSort Test.cs

(1 of 6)

**Fig. 25.10** | Class that creates an array filled with random integers. Provides a method to sort the array with merge sort. (Part 1 of 6).



## Outline

### InsertionSort Test.cs

(2 of 6)

```

21 // calls recursive SortArray method to begin merge sorting
22 public void sort()
23 {
24     SortArray( 0, data.Length - 1 ); // sort entire array
25 } // end method sort
26
27 // splits array, sorts subarrays and merges subarrays into sorted array
28 private void SortArray( int low, int high )
29 {
30     // test base case; size of array equals 1
31     if ( ( high - low ) >= 1 ) // if not base case
32     {
33         int middle1 = ( low + high ) / 2; // calculate middle of array
34         int middle2 = middle1 + 1; // calculate next element over
35
36         // output split step
37         Console.WriteLine( "split: " + Subarray( low, high ) );
38         Console.WriteLine( " " + Subarray( low, middle1 ) );
39         Console.WriteLine( " " + Subarray( middle2, high ) );
40         Console.WriteLine();
41

```

Calling method  
SortArray with the  
bounds of the entire array.

Testing whether the  
division has reached the  
base case.

**Fig. 25.10** | Class that creates an array filled with random integers. Provides a method to sort the array with merge sort. (Part 2 of 6).



## Outline

```

42     // split array in half; sort each half (recursive calls)
43     SortArray( low, middle1 ); // first half of array
44     SortArray( middle2, high ); // second half of array
45
46     // merge two sorted arrays after split calls return
47     Merge( low, middle1, middle2, high );
48 } // end if
49 } // end method SortArray
50
51 // merge two sorted subarrays into one sorted subarray
52 private void Merge( int left, int middle1, int middle2, int right )
53 {
54     int leftIndex = left; // index into left subarray
55     int rightIndex = middle2; // index into right subarray
56     int combinedIndex = left; // index into temporary working array
57     int[] combined = new int[ data.Length ]; // working array
58
59     // output two subarrays before merging
60     Console.WriteLine( "merge: " + Subarray( left, middle1 ) );
61     Console.WriteLine( " " + Subarray( middle2, right ) );
62

```

### InsertionSort Test.cs

(3 of 6)

Recursively calling  
SortArray on the first  
half of the array.

Recursively calling  
Sort-Array on the  
second half of the array.

Calling method Merge on  
the two sorted halves of  
the array.

**Fig. 25.10** | Class that creates an array filled with random integers. Provides a method to sort the array with merge sort. (Part 3 of 6).



## Outline

```

63 // merge arrays until reaching end of either
64 while ( leftIndex <= middle1 && rightIndex <= right )
65 {
66     // place smaller of two current elements into result
67     // and move to next space in arrays
68     if ( data[ leftIndex ] <= data[ rightIndex ] )
69         combined[ combinedIndex++ ] = data[ leftIndex++ ];
70     else
71         combined[ combinedIndex++ ] = data[ rightIndex++ ];
72 } // end while
73
74 // if left array is empty
75 if ( leftIndex == middle2 )
76     // copy in rest of right array
77     while ( rightIndex <= right )
78         combined[ combinedIndex++ ] = data[ rightIndex++ ];
79 else // right array is empty
80     // copy in rest of left array
81     while ( leftIndex <= middle1 )
82         combined[ combinedIndex++ ] = data[ leftIndex++ ];
83

```

### InsertionSort Test.cs

(4 of 6)

This loop repeats until a sorted array is created by merging two arrays.

Finishing the combined array once the left or right array has been completely integrated.

**Fig. 25.10** | Class that creates an array filled with random integers. Provides a method to sort the array with merge sort. (Part 4 of 6).



OutlineInsertionSort  
Test.cs

(5 of 6)

```

84 // copy values back into original array
85 for ( int i = left; i <= right; i++ )
86     data[ i ] = combined[ i ];
87
88 // output merged array
89 Console.WriteLine( " " + Subarray( left, right ) );
90 Console.WriteLine();
91 } // end method Merge
92
93 // method to output certain values in array
94 public string Subarray( int low, int high )
95 {
96     string temporary = string.Empty;
97
98     // output spaces for alignment
99     for ( int i = 0; i < low; i++ )
100         temporary += " ";
101
102     // output elements left in array
103     for ( int i = low; i <= high; i++ )
104         temporary += " " + data[ i ];

```

**Fig. 25.10** | Class that creates an array filled with random integers. Provides a method to sort the array with merge sort. (Part 5 of 6).



## Outline

### InsertionSort Test.cs

(6 of 6)

```
105
106     return temporary;
107 } // end method Subarray
108
109 // method to output values in array
110 public override string ToString()
111 {
112     return Subarray( 0, data.Length - 1 );
113 } // end method ToString
114} // end class MergeSort
```

**Fig. 25.10** | Class that creates an array filled with random integers. Provides a method to sort the array with merge sort. (Part 6 of 6).



## 25.3 Sorting Algorithms (Cont.)

- The first call to method `SortArray` results in two recursive calls with subarrays each half the size of the original array, and a single call to method `Merge`.
- The two calls to method `SortArray` result in four more recursive calls to `SortArray`.
- The total number of comparisons is  $(n/2 - 1) + (n/2 - 1) = n - 2$ , which is  $O(n)$ .
- Each level splits the size of the arrays in half, so doubling the size of the array requires only one more level. This results in a total efficiency of  $O(n \log n)$ .



## Outline

### MergeSortTest.cs

(1 of 3)

```
1 // Fig. 25.11: MergeSortTest.cs
2 // Testing the merge sort class.
3 using System;
4
5 public class MergeSortTest
6 {
7     public static void Main( string[] args )
8     {
9         // create object to perform merge sort
10        MergeSort sortArray = new MergeSort( 10 );
11
12        // print unsorted array
13        Console.WriteLine( "Unsorted: {0}\n", sortArray );
14
15        sortArray.Sort(); // sort array
16
17        // print sorted array
18        Console.WriteLine( "Sorted: {0}", sortArray );
19    } // end Main
20 } // end class MergeSortTest
```

**Fig. 25.11** | Testing the merge sort class. (Part 1 of 3).



Outline

```

Unsorted:  36 38 81 93 85 72 31 11 33 74
split:     36 38 81 93 85 72 31 11 33 74
           36 38 81 93 85
                72 31 11 33 74
split:     36 38 81 93 85
           36 38 81
                93 85
split:     36 38 81
           36 38
                81
split:     36 38
           36
                38
merge:     36
           38
           36 38
merge:     36 38
           36 38 81
           36 38 81
split:     93 85
           93
                85
merge:     93
           85
           85 93
merge:     36 38 81
           36 38 81 85 93
           36 38 81 85 93

```

*(continued on next page...)***MergeSortTest.cs**

(2 of 3)

**Fig. 25.11** | Testing the merge sort class. (Part 2 of 3).

Outline*(continued from previous page...)*

```

split:      72 31 11 33 74
            72 31 11
                33 74

```

```

split:      72 31 11
            72 31
                11

```

```

split:      72 31
            72
                31

```

```

merge:      72
            31
            31 72

```

```

merge:      31 72
            11
            11 31 72

```

```

split:      33 74
            33
                74

```

```

merge:      33
            74
            33 74

```

```

merge:      11 31 72
            33 74
            11 31 33 72 74

```

```

merge:      36 38 81 85 93
            11 31 33 72 74
            11 31 33 36 38 72 74 81 85 93

```

```

Sorted:    11 31 33 36 38 72 74 81 85 93

```

**MergeSortTest.cs**

(3 of 3)

**Fig. 25.11** | Testing the merge sort class. (Part 3 of 3).

## 25.3 Sorting Algorithms (Cont.)

- Figure 25.12 summarizes the searching and sorting algorithms covered in this book.

Algorithm	Location	Big O
<i>Searching Algorithms:</i>		
Linear Search	Section 25.2.1	$O(n)$
Binary Search	Section 25.2.2	$O(\log n)$
Recursive Linear Search	Exercise 25.8	$O(n)$
Recursive Binary Search	Exercise 25.9	$O(\log n)$
<i>Sorting Algorithms:</i>		
Selection Sort	Section 25.3.1	$O(n^2)$
Insertion Sort	Section 25.3.2	$O(n^2)$
Merge Sort	Section 25.3.3	$O(n \log n)$
Bubble Sort	Exercises 25.5–25.6	$O(n^2)$

**Fig. 25.12** | Searching and sorting algorithms with Big O values.



## 25.3 Sorting Algorithms (Cont.)

- Figure 25.13 lists the Big O values covered in this chapter, along with values for  $n$  to highlight growth rates.

$n =$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10000
1,000	3	1000	3000	$10^6$
1,000,000	6	1000000	6000000	$10^{12}$
1,000,000,000	9	1000000000	9000000000	$10^{18}$

**Fig. 25.13** | Number of comparisons for common Big O notations.

