

# Software Design 2

## SDN260S

### Searching, Sorting & Big O

*H. Mataifa*

Department of Electronic, Electrical  
and Computer Engineering

# Outline

- Searching algorithms
  - Linear search
  - Binary search
- Algorithm complexity/efficiency (Big O)
- Sorting algorithms
  - Selection sort
  - Insertion sort
  - Merge sort

# Searching And Sorting

- **Searching**: involves determining whether a value (**search key**) is present in some data collection; if present, location in the data is returned
- Two popular search algorithms:
  - **Linear search**: simple, but quite slow
  - **Binary search**: faster, but more complex
- **Sorting**: places data in some order (ascending/descending) (e.g. alphabetical ordering of strings, ascending ordering of numbers, etc.)
- Three common sorting algorithms:
  - **Selection sort**
  - **Insertion sort**
  - **Merge sort**: more efficient, but also more complex

# Searching Algorithms: Linear Search

## Linear search algorithm:

- Performs **sequential** search of array data for the **search key**:
  - Each array element (from first one) is tested against **search key**, until either the **search key** is found, or end of array is reached
  - If **search key** is found, index of the element is returned
  - If there are duplicate values in the array, index of only first element is returned

# Linear Search

```
1 // Fig. 19.2: LinearArray.java
2 // Class that contains an array of random integers and a method
3 // that will search that array sequentially.
4 import java.util.Random;
5 import java.util.Arrays;
6
7 public class LinearArray
8 {
9     private int[] data; // array of values
10    private static final Random generator = new Random();
11
12    // create array of given size and fill with random numbers
13    public LinearArray( int size )
14    {
15        data = new int[ size ]; // create space for array
16
17        // fill array with random ints in range 10-99
18        for ( int i = 0; i < size; i++ )
19            data[ i ] = 10 + generator.nextInt( 90 );
20    } // end LinearArray constructor
21
22    // perform a linear search on the data
23    public int linearSearch( int searchKey )
24    {
25        // loop through array sequentially
26        for ( int index = 0; index < data.length; index++ )
27            if ( data[ index ] == searchKey )
28                return index; // return index of integer
29
30        return -1; // integer was not found
31    } // end method linearSearch
32
33    // method to output values in array
34    public String toString()
35    {
36        return Arrays.toString( data );
37    } // end method toString
38 } // end class LinearArray
```

*Array of random numbers*

*Linear search of array elements*

# Linear Search (Test)

```
1 // Fig. 19.3: LinearSearchTest.java
2 // Sequentially searching an array for an item.
3 import java.util.Scanner;
4
5 public class LinearSearchTest
6 {
7     public static void main( String[] args )
8     {
9         // create Scanner object to input data
10        Scanner input = new Scanner( System.in );
11
12        int searchInt; // search key
13        int position; // location of search key in array
14
15        // create array and output it
16        LinearArray searchArray = new LinearArray( 10 );
17        System.out.println( searchArray + "\n" ); // print array
18
19        // get input from user
20        System.out.print(
21            "Please enter an integer value (-1 to quit): " );
22        searchInt = input.nextInt(); // read first int from user
23
24        // repeatedly input an integer; -1 terminates the program
25        while ( searchInt != -1 )
26        {
27            // perform linear search
28            position = searchArray.linearSearch( searchInt );
29
30            if ( position == -1 ) // integer was not found
31                System.out.println( "The integer " + searchInt +
32                    " was not found.\n" );
33            else // integer was found
34                System.out.println( "The integer " + searchInt +
35                    " was found in position " + position + ".\n" );
36
37            // get input from user
38            System.out.print(
39                "Please enter an integer value (-1 to quit): " );
40            searchInt = input.nextInt(); // read next int from user
41        } // end while
42    } // end main
43 } // end class LinearSearchTest
```

Scanner object to fetch input  
data from keyboard

Request user to input number  
to search for

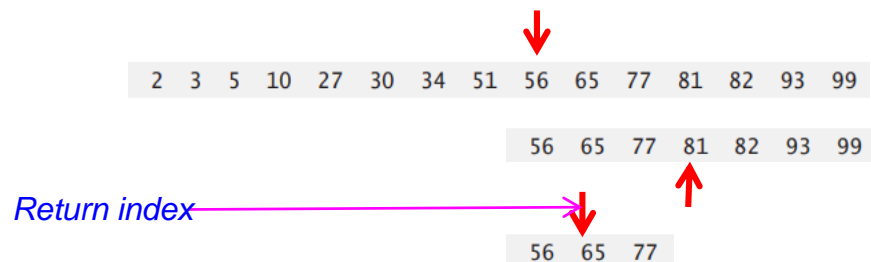
Random-number array to be  
searched for search key

Perform linear search and  
Output result

# Searching Algorithms: Binary Search

## Binary search algorithm:

- Searches array for **search key** by **bisecting array** into two halves:
  - First test of **search key** is against **middle element** of the array
  - Three possible outcomes:
    - **Match**: index is returned
    - **Search key less** than **middle element**: discard upper half of array, perform binary search on lower half
    - **Search key greater** than **middle element**: discard lower half of array, perform binary search on upper half
  - Repeat until either search key is found or search fails
- Binary search algorithm requires array to be sorted (typically in ascending order)



*Binary search for 65 in 15-element ordered array*

# Binary Search

```
1 // Fig. 19.4: BinaryArray.java
2 // Class that contains an array of random integers and a method
3 // that uses binary search to find an integer.
4 import java.util.Random;
5 import java.util.Arrays;
6
7 public class BinaryArray
8 {
9     private int[] data; // array of values
10    private static final Random generator = new Random();
11
12    // create array of given size and fill with random integers
13    public BinaryArray( int size )
14    {
15        data = new int[ size ]; // create space for array
16
17        // fill array with random ints in range 10-99
18        for ( int i = 0; i < size; i++ )
19            data[ i ] = 10 + generator.nextInt( 90 );
20
21        Arrays.sort( data );
22    } // end BinaryArray constructor
23
24    // perform a binary search on the data
25    public int binarySearch( int searchElement )
26    {
27        int low = 0; // low end of the search area
28        int high = data.length - 1; // high end of the search area
29        int middle = ( low + high + 1 ) / 2; // middle element
30        int location = -1; // return value; -1 if not found
31
32        do // loop to search for element
33        {
34            // print remaining elements of array
35            System.out.print( remainingElements( low, high ) );
36
37            // output spaces for alignment
38            for ( int i = 0; i < middle; i++ )
39                System.out.print( " " );
40            System.out.println( " * " ); // 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
```

*Sorted array construction*

*Binary search of sorted array*



# Binary Search

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

# Binary Search (Test)

```
1 // Fig. 19.5: BinarySearchTest.java
2 // Use binary search to locate an item in an array.
3 import java.util.Scanner;
4
5 public class BinarySearchTest
6 {
7     public static void main( String[] args )
8     {
9         // create Scanner object to input data
10        Scanner input = new Scanner( System.in );
11
12        int searchInt; // search key
13        int position; // location of search key in array
14
15        // create array and output it
16        BinaryArray searchArray = new BinaryArray( 15 );
17        System.out.println( searchArray );
18
19        // get input from user
20        System.out.print(
21            "Please enter an integer value (-1 to quit): " );
22        searchInt = input.nextInt(); // read an int from user
23        System.out.println();
24
25        // repeatedly input an integer; -1 terminates the program
26        while ( searchInt != -1 )
27        {
28            // use binary search to try to find integer
29            position = searchArray.binarySearch( searchInt );
30
31            // return value of -1 indicates integer was not found
32            if ( position == -1 )
33                System.out.println( "The integer " + searchInt +
34                    " was not found.\n" );
35            else
36                System.out.println( "The integer " + searchInt +
37                    " was found in position " + position + ".\n" );
38
39            // get input from user
40            System.out.print(
41                "Please enter an integer value (-1 to quit): " );
42            searchInt = input.nextInt(); // read an int from user
43            System.out.println();
44        } // end while
45    } // end main
46 } // end class BinarySearchTest
```

## Algorithm Complexity/Efficiency (Section 19.2.1)

- Search algorithms all eventually accomplish the same goal- find an element that matches a given **search key**, if one does exist
  - *Main difference is amount of effort required to complete the search*
- **Big O notation**: indicates the **worst-case run-time** for an algorithm-that is, how hard an algorithm may have to work to solve a problem
- For searching and sorting algorithms (and for many other computerized algorithmic tasks), *amount of effort usually depends on how many data elements there are*
  - **O(1)** algorithms: effort to complete the task **independent** of the number of data elements (e.g. comparing two specific elements in a given array); the algorithm is said to have **constant runtime**
  - **O(n)** algorithms: effort to complete the task **proportional to number of data elements** (more comparisons required as n grows) (e.g. Linear search algorithm is O(n)); the algorithm is said to have **linear runtime**
  - **O(n<sup>2</sup>)** algorithm: effort to complete the task **proportional to square of number of data elements** (e.g. algorithm required to test whether an array has any duplicates); algorithm said to have **quadratic runtime**
- **Big O** is concerned with how an algorithm's runtime grows in relation to number of items processed; **designing efficient algorithms** has the goal of achieving a favourable **Big O** (leads to faster, more efficient programs)

# Sorting Algorithms

- **Sorting**: places data in some order (ascending/descending) (e.g. alphabetical ordering of strings, ascending ordering of numbers, etc.)
- A variety of sorting algorithms, all which achieve the same objective, only differ in complexity and efficiency:
  - Selection sort
  - Insertion sort
  - Merge sort
- Choice of algorithm affects run-time and memory usage of program (but result should be the same)
- Compromise between ease of programming (simplicity) and efficiency (execution time, memory requirement)
  - Selection sort and insertion sort simple (i.e. easy to program), but inefficient
  - Merge sort more efficient, but more complex

# Sorting Algorithms: Selection Sort

## Selection sort algorithm:

- Simple, but inefficient algorithm
- Procedure (ascending order):
  - Iterate through all array elements, find smallest element, swap with first element
  - Iterate through all array elements except first one, find smallest element, swap with second element
  - Repeat above until all array elements are sorted (last iteration swaps last element with second-to-last one)
- After  $i^{th}$  iteration, first  $i$  elements are sorted in ascending order
- Selection sort algorithm complexity is  $O(n^2)$

Original	34	56	4	10	77	51	93	30	5	52	Third element swapped with first
Iter. 1	4	56	34	10	77	51	93	30	5	52	Ninth element swapped with second
Iter. 2	4	5	34	10	77	51	93	30	56	52	Fourth element swapped with third
Iter. 3	4	5	10	34	77	51	93	30	56	52	Eighth element swapped with fourth
Iter. n-1	4	5	10	30	34	51	52	56	77	93	

# Selection Sort Algorithm

```
1 // Fig. 19.6: SelectionSort.java
2 // Class that creates an array filled with random integers.
3 // Provides a method to sort the array with selection sort.
4 import java.util.Arrays;
5 import java.util.Random;
6
7 public class SelectionSort
8 {
9     private int[] data; // array of values
10    private static final Random generator = new Random();
11
12    // create array of given size and fill with random integers
13    public SelectionSort( int size )
14    {
15        data = new int[ size ]; // create space for array
16
17        // fill array with random ints in range 10-99
18        for ( int i = 0; i < size; i++ )
19            data[ i ] = 10 + generator.nextInt( 90 );
20    } // end SelectionSort constructor
21
22    // sort array using selection sort
23    public void sort()
24    {
25        int smallest; // index of smallest element
26
27        // loop over data.length - 1 elements
28        for ( int i = 0; i < data.length - 1; i++ )
29        {
30            smallest = i; // first index of remaining array
31
32            // loop to find index of smallest element
33            for ( int index = i + 1; index < data.length; index++ )
34                if ( data[ index ] < data[ smallest ] )
35                    smallest = index;
36
37            swap( i, smallest ); // swap smallest element into position
38            printPass( i + 1, smallest ); // output pass of algorithm
39        } // end outer for
40    } // end method sort
41
42    // swap two elements in array
43    private void swap( int i, int j )
44    {
45        int temp = data[ i ];
46        data[ i ] = data[ j ];
47        data[ j ] = temp;
48    }
49
50    // print array
51    private void printPass( int pass, int index )
52    {
53        System.out.print( "Pass " + pass + ": " );
54        for ( int i = 0; i < data.length; i++ )
55            System.out.print( data[ i ] + " " );
56        System.out.println();
57    }
58
59    // print array
60    private void printArray()
61    {
62        System.out.print( "Array: " );
63        for ( int i = 0; i < data.length; i++ )
64            System.out.print( data[ i ] + " " );
65        System.out.println();
66    }
67
68    // test the program
69    public static void main( String[] args )
70    {
71        SelectionSort sorter = new SelectionSort( 10 );
72        sorter.printArray();
73        sorter.sort();
74        sorter.printArray();
75    }
76}
```

# Selection Sort Algorithm

```
41
42 // helper method to swap values in two elements
43 public void swap( int first, int second )
44 {
45     int temporary = data[ first ]; // store first in temporary
46     data[ first ] = data[ second ]; // replace first with second
47     data[ second ] = temporary; // put temporary in second
48 } // end method swap
49
50 // print a pass of the algorithm
51 public void printPass( int pass, int index )
52 {
53     System.out.print( String.format( "after pass %2d: ", pass ) );
54
55     // output elements till selected item
56     for ( int i = 0; i < index; i++ )
57         System.out.print( data[ i ] + " " );
58
59     System.out.print( data[ index ] + "* " ); // indicate swap
60
61     // finish outputting array
62     for ( int i = index + 1; i < data.length; i++ )
63         System.out.print( data[ i ] + " " );
64
65     System.out.print( "\n" ); // for alignment
66
67     // indicate amount of array that is sorted
68     for ( int j = 0; j < pass; j++ )
69         System.out.print( "-- " );
70     System.out.println( "\n" ); // add newline
71 } // end method printPass
72
73 // method to output values in array
74 public String toString()
75 {
76     return Arrays.toString( data );
77 } // end method toString
78 } // end class SelectionSort
```

# Selection Sort Test Program

```
1 // Fig. 19.7: SelectionSortTest.java
2 // Testing the selection sort class.
3
4 public class SelectionSortTest
5 {
6     public static void main( String[] args )
7     {
8         // create object to perform selection sort
9         SelectionSort sortArray = new SelectionSort( 10 );
10
11         System.out.println( "Unsorted array:" );
12         System.out.println( sortArray + "\n" ); // print unsorted array
13
14         sortArray.sort(); // sort array
15
16         System.out.println( "Sorted array:" );
17         System.out.println( sortArray ); // print sorted array
18     } // end main
19 } // end class SelectionSortTest
```



# Sorting Algorithms: Insertion Sort

## Insertion sort algorithm:

- Also simple, but inefficient algorithm
- Procedure (ascending order):
  - First iteration compares second element with first, swaps if less than first
  - Second iteration compares third element with first two, inserts into proper location relative to the first two, so that first three elements are sorted relative to one another
  - Repeat above until all array elements are sorted
- After  $i^{th}$  iteration, first  $i$  elements are sorted in ascending order
- Insertion sort algorithm complexity is  $O(n^2)$

Original  
Iter. 1

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

2nd element compared with 1st; *no swap*

Iter. 2

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

3rd element compared with 1st two, *swapped*

Iter. 3

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

4th element compared with 1<sup>st</sup> three, *swapped*

# Insertion Sort Algorithm

```
1 // Fig. 19.8: InsertionSort.java
2 // Class that creates an array filled with random integers.
3 // Provides a method to sort the array with insertion sort.
4 import java.util.Arrays;
5 import java.util.Random;
6
7 public class InsertionSort
8 {
9     private int[] data; // array of values
10    private static final Random generator = new Random();
11
12    // create array of given size and fill with random integers
13    public InsertionSort( int size )
14    {
15        data = new int[ size ]; // create space for array
16
17        // fill array with random ints in range 10-99
18        for ( int i = 0; i < size; i++ )
19            data[ i ] = 10 + generator.nextInt( 90 );
20    } // end InsertionSort constructor
21
22    // sort array using insertion sort
23    public void sort()
24    {
25        int insert; // temporary variable to hold element to insert
26
27        // loop over data.length - 1 elements
28        for ( int next = 1; next < data.length; next++ )
29        {
30            // store value in current element
31            insert = data[ next ];
32
33            // initialize location to place element
34            int moveItem = next;
35
36            // search for place to put current element
37            while ( moveItem > 0 && data[ moveItem - 1 ] > insert )
38            {
39                // shift element right one slot
40                data[ moveItem ] = data[ moveItem - 1 ];
41                moveItem--;
42            } // end while
43
44            data[ moveItem ] = insert; // place inserted element
45            printPass( next, moveItem ); // output pass of algorithm
46        } // end for
47    } // end method sort
```

# Insertion Sort Algorithm

```
48
49 // print a pass of the algorithm
50 public void printPass( int pass, int index )
51 {
52     System.out.print( String.format( "after pass %2d: ", pass ) );
53
54     // output elements till swapped item
55     for ( int i = 0; i < index; i++ )
56         System.out.print( data[ i ] + " " );
57
58     System.out.print( data[ index ] + "* " ); // indicate swap
59
60     // finish outputting array
61     for ( int i = index + 1; i < data.length; i++ )
62         System.out.print( data[ i ] + " " );
63
64     System.out.print( "\n" ); // for alignment
65
66     // indicate amount of array that is sorted
67     for( int i = 0; i <= pass; i++ )
68         System.out.print( "-- " );
69     System.out.println( "\n" ); // add newline
70 } // end method printPass
71
72 // method to output values in array
73 public String toString()
74 {
75     return Arrays.toString( data );
76 } // end method toString
77 } // end class InsertionSort
```

# Insertion Sort Test Program

```
1  // Fig. 19.9: InsertionSortTest.java
2  // Testing the insertion sort class.
3
4  public class InsertionSortTest
5  {
6      public static void main( String[] args )
7      {
8          // create object to perform insertion sort
9          InsertionSort sortArray = new InsertionSort( 10 );
10
11          System.out.println( "Unsorted array:" );
12          System.out.println( sortArray + "\n" ); // print unsorted array
13
14          sortArray.sort(); // sort array
15
16          System.out.println( "Sorted array:" );
17          System.out.println( sortArray ); // print sorted array
18      } // end main
19  } // end class InsertionSortTest
```

# Sorting Algorithms: Merge Sort

## Merge sort algorithm:

- Efficient, but more complex sorting algorithm (relative to selection, insertion sort)
- Procedure:
  - Sorts an array by splitting it into two (nearly) equal-sized subarrays, sorting each subarray, then merging them into one large array
  - Uses recursion to sort each subarray (i.e. each subarray is in turn split into subarrays, and merge sort is applied to each subsequent subarray)
  - Base case is a subarray with one element, which causes the method to return the element
  - Recursion step is what splits the array into subarray (given that the array has more than one element)
- Merge sort algorithm complexity is  $O(n \log n)$  (better than  $O(n^2)$ )

# Merge Sort Algorithm

```
1 // Fig. 19.10: MergeSort.java
2 // Class creates an array filled with random integers.
3 // Provides a method to sort the array with merge sort.
4 import java.util.Random;
5
6 public class MergeSort
7 {
8     private int[] data; // array of values
9     private static final 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 ] = 10 + generator.nextInt( 90 );
19     } // end MergeSort constructor
20
21     // calls recursive split method to begin merge sorting
22     public void sort()
23     {
24         sortArray( 0, data.length - 1 ); // split 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             System.out.println( "split:  " + subarray( low, high ) );
38             System.out.println( "        " + subarray( low, middle1 ) );
39             System.out.println( "        " + subarray( middle2, high ) );
40             System.out.println();
41
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 }
```

# Merge Sort Algorithm

```
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     System.out.println( "merge: " + subarray( left, middle1 ) );
61     System.out.println( " " + subarray( middle2, right ) );
62
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
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     System.out.println( " " + subarray( left, right ) );
90     System.out.println();
91 } // end method merge
92
93 // method to output certain values in array
94 public String subarray( int low, int high )
95 {
96     StringBuilder temporary = new StringBuilder();
97
98     // output spaces for alignment
99     for ( int i = 0; i < low; i++ )
100         temporary.append( " " );
101
102     // output elements left in array
103     for ( int i = low; i <= high; i++ )
104         temporary.append( " " + data[ i ] );
105
106     return temporary.toString();
107 } // end method subarray
108
109 // method to output values in array
110 public String toString()
111 {
112     return subarray( 0, data.length - 1 );
113 } // end method toString
114 } // end class MergeSort
```

# Merge Sort Algorithm Test Program

```
1 // Figure 16.11: MergeSortTest.java
2 // Testing the merge sort class.
3
4 public class MergeSortTest
5 {
6     public static void main( String[] args )
7     {
8         // create object to perform merge sort
9         MergeSort sortArray = new MergeSort( 10 );
10
11         // print unsorted array
12         System.out.println( "Unsorted:" + sortArray + "\n" );
13
14         sortArray.sort(); // sort array
15
16         // print sorted array
17         System.out.println( "Sorted:  " + sortArray );
18     } // end main
19 } // end class MergeSortTest
```



## Exercises

**19.8** (*Recursive Linear Search*) Modify Fig. 19.2 to use recursive method `recursiveLinearSearch` to perform a linear search of the array. The method should receive the search key and starting index as arguments. If the search key is found, return its index in the array; otherwise, return -1. Each call to the recursive method should check one index in the array.

**19.9** (*Recursive Binary Search*) Modify Fig. 19.4 to use recursive method `recursiveBinarySearch` to perform a binary search of the array. The method should receive the search key, starting index and ending index as arguments. If the search key is found, return its index in the array. If the search key is not found, return -1.

## Program code for exercise 19.8

```
package com.mycompany.classon230918;

import java.util.Random;
import java.util.Scanner;
import java.util.Arrays;

public class LinearSearch {

    private static int[] array;

    public static void main(String[] args) {

        Random generator = new Random();
        Scanner input = new Scanner(System.in);
        array = new int[10];

        // generate 10 random integers from 1 to 29:
        for (int i = 0; i < 10; i++)
            array[i] = 1 + generator.nextInt(19);

        // Request user to enter element to search for:
        System.out.println("\nPlease enter the integer to search for: ");
        int searchKey = input.nextInt();

        // Search the array for the element:
        int iterative_index = iterativeLinearSearch(searchKey);
        int recursive_index = recursiveLinearSearch(searchKey, 0);
    }
}
```

## Program code for exercise 19.8

```
// Search for the element using the iterative method:
if (iterative_index == -1)
    System.out.printf("\nUsing the iterative method, %d was not found in array %s", searchKey,
        Arrays.toString(array));
else
    System.out.printf("\nUsing the iterative method, %d was found in array %s at position %d", searchKey,
        Arrays.toString(array), iterative_index);

// Search for the element using the recursive method:
if (recursive_index == -1)
    System.out.printf("\nUsing the recursive method, %d was not found in array %s", searchKey,
        Arrays.toString(array));
else
    System.out.printf("\nUsing the recursive method, %d was found in array %s at position %d", searchKey,
        Arrays.toString(array), recursive_index);

}

// Define iterative linear search algorithm:
public static int iterativeLinearSearch(int searchElement)
{
    int position = -1;
    for (int i = 0; i < array.length; i++)
    {
        if (array[i] == searchElement)
            return i;
    }

    return position;
}
```

## Program code for exercise 19.8

```
// Define recursive linear search algorithm:
public static int recursiveLinearSearch(int searchElement, int index)
{
    if (index == array.length-1 && array[index] != searchElement)
        return -1;

    if (array[index] == searchElement)
        return index;
    else
        return recursiveLinearSearch(searchElement, index + 1);
}

}
```