Cape Peninsula University of Technology

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 collection 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

```
// Fig. 19.2: LinearArray.java
// Class that contains an array of random integers and a method
// that will search that array sequentially.
import java.util.Random;
import java.util.Arrays;

public class LinearArray
{
   private int[] data; // array of values
   private static final Random generator = new Random();

   // create array of given size and fill with random numbers
   public LinearArray( int size )
   {
      data = new int[ size ]; // create space for array

      // fill array with random ints in range 10-99
      for ( int i = 0; i < size; i++ )
         data[ i ] = 10 + generator.nextInt( 90 );
   } // end LinearArray constructor

   // perform a linear search on the data
   public int linearSearch( int searchKey )
   {
      // loop through array sequentially
      for ( int index = 0; index < data.length; index++ )
         if ( data[ index ] == searchKey )
            return index; // return index of integer

      return -1; // integer was not found
   } // end method linearSearch

   // method to output values in array
   public String toString()
   {
      return Arrays.toString( data );
   } // end method toString
} // end class LinearArray
```

Array of random numbers

Linear search of array elements

Linear Search (Test)

```java
// Fig. 19.3: LinearSearchTest.java
// Sequentially searching an array for an item.
import java.util.Scanner;

public class LinearSearchTest
{
   public static void main( String[] args )
   {
      // create Scanner object to input data
      Scanner input = new Scanner( System.in );

      int searchInt; // search key
      int position; // location of search key in array

      // create array and output it
      LinearArray searchArray = new LinearArray( 10 );
      System.out.println( searchArray + "\n" ); // print array

      // get input from user
      System.out.print(
         "Please enter an integer value (-1 to quit): " );
      searchInt = input.nextInt(); // read first int from user

      // repeatedly input an integer; -1 terminates the program
      while ( searchInt != -1 )
      {
         // perform linear search
         position = searchArray.linearSearch( searchInt );

         if ( position == -1 ) // integer was not found
            System.out.println( "The integer " + searchInt +
               " was not found.\n" );
         else // integer was found
            System.out.println( "The integer " + searchInt +
               " was found in position " + position + ".\n" );

         // get input from user
         System.out.print(
            "Please enter an integer value (-1 to quit): " );
         searchInt = input.nextInt(); // read next int from user
      } // end while
   } // end main
} // end class LinearSearchTest
```

Scanner object to fetch input data from keyboard

Random-number array to be searched for search key

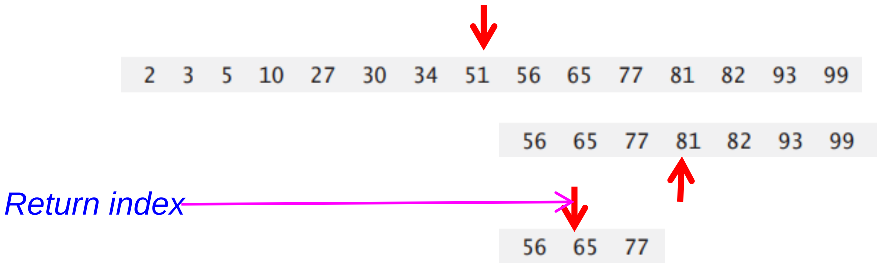
Request user to input number to search for

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

```java
// Fig. 19.4: BinaryArray.java
// Class that contains an array of random integers and a method
// that uses binary search to find an integer.
import java.util.Random;
import java.util.Arrays;

public class BinaryArray
{
   private int[] data; // array of values
   private static final Random generator = new Random();

   // create array of given size and fill with random integers
   public BinaryArray( int size )
   {
      data = new int[ size ]; // create space for array

      // fill array with random ints in range 10-99
      for ( int i = 0; i < size; i++ )
         data[ i ] = 10 + generator.nextInt( 90 );

      Arrays.sort( data );
   } // end BinaryArray constructor

   // perform a binary search on the data
   public int binarySearch( int searchElement )
   {
      int low = 0; // low end of the search area
      int high = data.length - 1; // high end of the search area
      int middle = ( low + high + 1 ) / 2; // middle element
      int location = -1; // return value; -1 if not found

      do // loop to search for element
      {
         // print remaining elements of array
         System.out.print( remainingElements( low, high ) );

         // output spaces for alignment
         for ( int i = 0; i < middle; i++ )
            System.out.print( "   " );
         System.out.println( " * " ); // indicate current middle

         // if the element is found at the middle
         if ( searchElement == data[ middle ] )
            location = middle; // location is the current middle

         // middle element is too high
         else if ( searchElement < data[ middle ] )
            high = middle - 1; // eliminate the higher half
         else  // middle element is too low
            low = middle + 1; // eliminate the lower half

         middle = ( low + high + 1 ) / 2; // recalculate the middle
      } while ( ( low <= high ) && ( location == -1 ) );

      return location; // return location of search key
   } // end method binarySearch
```

Sorted array construction

Binary search of sorted array

Binary Search

```java
   // method to output certain values in array
   public String remainingElements( int low, int high )
   {
      StringBuilder temporary = new StringBuilder();

      // output spaces for alignment
      for ( int i = 0; i < low; i++ )
         temporary.append( "   " );

      // output elements left in array
      for ( int i = low; i <= high; i++ )
         temporary.append( data[ i ] + " " );

      temporary.append( "\n" );
      return temporary.toString();
   } // end method remainingElements

   // method to output values in array
   public String toString()
   {
      return remainingElements( 0, data.length - 1 );
   } // end method toString
} // end class BinaryArray
```

Binary Search (Test)

```java
// Fig. 19.5: BinarySearchTest.java
// Use binary search to locate an item in an array.
import java.util.Scanner;

public class BinarySearchTest
{
   public static void main( String[] args )
   {
      // create Scanner object to input data
      Scanner input = new Scanner( System.in );

      int searchInt; // search key
      int position; // location of search key in array

      // create array and output it
      BinaryArray searchArray = new BinaryArray( 15 );
      System.out.println( searchArray );

      // get input from user
      System.out.print(
         "Please enter an integer value (-1 to quit): " );
      searchInt = input.nextInt(); // read an int from user
      System.out.println();

      // repeatedly input an integer; -1 terminates the program
      while ( searchInt != -1 )
      {
         // use binary search to try to find integer
         position = searchArray.binarySearch( searchInt );

         // return value of -1 indicates integer was not found
         if ( position == -1 )
            System.out.println( "The integer " + searchInt +
               " was not found.\n" );
         else
            System.out.println( "The integer " + searchInt +
               " was found in position " + position + ".\n" );

         // get input from user
         System.out.print(
            "Please enter an integer value (-1 to quit): " );
         searchInt = input.nextInt(); // read an int from user
         System.out.println();
      } // end while
   } // end main
} // 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^2)** 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

```java
// Fig. 19.6: SelectionSort.java
// Class that creates an array filled with random integers.
// Provides a method to sort the array with selection sort.
import java.util.Arrays;
import java.util.Random;

public class SelectionSort
{
   private int[] data; // array of values
   private static final Random generator = new Random();

   // create array of given size and fill with random integers
   public SelectionSort( int size )
   {
      data = new int[ size ]; // create space for array

      // fill array with random ints in range 10-99
      for ( int i = 0; i < size; i++ )
         data[ i ] = 10 + generator.nextInt( 90 );
   } // end SelectionSort constructor

   // sort array using selection sort
   public void sort()
   {
      int smallest; // index of smallest element

      // loop over data.length - 1 elements
      for ( int i = 0; i < data.length - 1; i++ )
      {
         smallest = i; // first index of remaining array

         // loop to find index of smallest element
         for ( int index = i + 1; index < data.length; index++ )
            if ( data[ index ] < data[ smallest ] )
               smallest = index;

         swap( i, smallest ); // swap smallest element into position
         printPass( i + 1, smallest ); // output pass of algorithm
      } // end outer for
   } // end method sort
```

Selection Sort Algorithm

```java

    // helper method to swap values in two elements
    public void swap( int first, int second )
    {
       int temporary = data[ first ]; // store first in temporary
       data[ first ] = data[ second ]; // replace first with second
       data[ second ] = temporary; // put temporary in second
    } // end method swap

    // print a pass of the algorithm
    public void printPass( int pass, int index )
    {
       System.out.print( String.format( "after pass %2d: ", pass ) );

       // output elements till selected item
       for ( int i = 0; i < index; i++ )
          System.out.print( data[ i ] + "  " );

       System.out.print( data[ index ] + "* " ); // indicate swap

       // finish outputting array
       for ( int i = index + 1; i < data.length; i++ )
          System.out.print( data[ i ] + "  " );

       System.out.print( "\n               " ); // for alignment

       // indicate amount of array that is sorted
       for ( int j = 0; j < pass; j++ )
          System.out.print( "--  " );
       System.out.println( "\n" ); // add endline
    } // end method printPass

    // method to output values in array
    public String toString()
    {
       return Arrays.toString( data );
    } // end method toString
} // end class SelectionSort
```

Selection Sort Test Program

```
// Fig. 19.7: SelectionSortTest.java
// Testing the selection sort class.

public class SelectionSortTest
{
   public static void main( String[] args )
   {
      // create object to perform selection sort
      SelectionSort sortArray = new SelectionSort( 10 );

      System.out.println( "Unsorted array:" );
      System.out.println( sortArray + "\n" ); // print unsorted array

      sortArray.sort(); // sort array

      System.out.println( "Sorted array:" );
      System.out.println( sortArray ); // print sorted array
   } // end main
} // 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 1st three, swapped*

Insertion Sort Algorithm

```
// Fig. 19.8: InsertionSort.java
// Class that creates an array filled with random integers.
// Provides a method to sort the array with insertion sort.
import java.util.Arrays;
import java.util.Random;

public class InsertionSort
{
   private int[] data; // array of values
   private static final Random generator = new Random();

   // create array of given size and fill with random integers
   public InsertionSort( int size )
   {
      data = new int[ size ]; // create space for array

      // fill array with random ints in range 10-99
      for ( int i = 0; i < size; i++ )
         data[ i ] = 10 + generator.nextInt( 90 );
   } // end InsertionSort constructor

   // sort array using insertion sort
   public void sort()
   {
      int insert; // temporary variable to hold element to insert

      // loop over data.length - 1 elements
      for ( int next = 1; next < data.length; next++ )
      {
         // store value in current element
         insert = data[ next ];

         // initialize location to place element
         int moveItem = next;

         // search for place to put current element
         while ( moveItem > 0 && data[ moveItem - 1 ] > insert )
         {
            // shift element right one slot
            data[ moveItem ] = data[ moveItem - 1 ];
            moveItem--;
         } // end while

         data[ moveItem ] = insert; // place inserted element
         printPass( next, moveItem ); // output pass of algorithm
      } // end for
   } // end method sort
```

Insertion Sort Algorithm

```java

    // print a pass of the algorithm
    public void printPass( int pass, int index )
    {
       System.out.print( String.format( "after pass %2d: ", pass ) );

       // output elements till swapped item
       for ( int i = 0; i < index; i++ )
          System.out.print( data[ i ] + "  " );

       System.out.print( data[ index ] + "* " ); // indicate swap

       // finish outputting array
       for ( int i = index + 1; i < data.length; i++ )
          System.out.print( data[ i ] + "  " );

       System.out.print( "\n                  " ); // for alignment

       // indicate amount of array that is sorted
       for( int i = 0; i <= pass; i++ )
          System.out.print( "--  " );
       System.out.println( "\n" ); // add endline
    } // end method printPass

    // method to output values in array
    public String toString()
    {
       return Arrays.toString( data );
    } // end method toString
 } // end class InsertionSort
```

Insertion Sort Test Program

```java
// Fig. 19.9: InsertionSortTest.java
// Testing the insertion sort class.

public class InsertionSortTest
{
   public static void main( String[] args )
   {
      // create object to perform insertion sort
      InsertionSort sortArray = new InsertionSort( 10 );

      System.out.println( "Unsorted array:" );
      System.out.println( sortArray + "\n" ); // print unsorted array

      sortArray.sort(); // sort array

      System.out.println( "Sorted array:" );
      System.out.println( sortArray ); // print sorted array
   } // end main
} // 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**(nlogn) (better than $O(n^2)$)

Merge Sort Algorithm

```java
// Fig. 19.10: MergeSort.java
// Class creates an array filled with random integers.
// Provides a method to sort the array with merge sort.
import java.util.Random;

public class MergeSort
{
   private int[] data; // array of values
   private static final Random generator = new Random();

   // create array of given size and fill with random integers
   public MergeSort( int size )
   {
      data = new int[ size ]; // create space for array

      // fill array with random ints in range 10-99
      for ( int i = 0; i < size; i++ )
         data[ i ] = 10 + generator.nextInt( 90 );
   } // end MergeSort constructor

   // calls recursive split method to begin merge sorting
   public void sort()
   {
      sortArray( 0, data.length - 1 ); // split entire array
   } // end method sort

   // splits array, sorts subarrays and merges subarrays into sorted array
   private void sortArray( int low, int high )
   {
      // test base case; size of array equals 1
      if ( ( high - low ) >= 1 ) // if not base case
      {
         int middle1 = ( low + high ) / 2; // calculate middle of array
         int middle2 = middle1 + 1; // calculate next element over

         // output split step
         System.out.println( "split:   " + subarray( low, high ) );
         System.out.println( "         " + subarray( low, middle1 ) );
         System.out.println( "         " + subarray( middle2, high ) );
         System.out.println();

         // split array in half; sort each half (recursive calls)
         sortArray( low, middle1 ); // first half of array
         sortArray( middle2, high ); // second half of array

         // merge two sorted arrays after split calls return
         merge ( low, middle1, middle2, high );
      } // end if
   } // end method sortArray
```

Merge Sort Algorithm

```java
   // merge two sorted subarrays into one sorted subarray
   private void merge( int left, int middle1, int middle2, int right )
   {
      int leftIndex = left; // index into left subarray
      int rightIndex = middle2; // index into right subarray
      int combinedIndex = left; // index into temporary working array
      int[] combined = new int[ data.length ]; // working array

      // output two subarrays before merging
      System.out.println( "merge:   " + subarray( left, middle1 ) );
      System.out.println( "         " + subarray( middle2, right ) );

      // merge arrays until reaching end of either
      while ( leftIndex <= middle1 && rightIndex <= right )
      {
         // place smaller of two current elements into result
         // and move to next space in arrays
         if ( data[ leftIndex ] <= data[ rightIndex ] )
            combined[ combinedIndex++ ] = data[ leftIndex++ ];
         else
            combined[ combinedIndex++ ] = data[ rightIndex++ ];
      } // end while

      // if left array is empty
      if ( leftIndex == middle2 )
         // copy in rest of right array
         while ( rightIndex <= right )
            combined[ combinedIndex++ ] = data[ rightIndex++ ];
      else // right array is empty
         // copy in rest of left array
         while ( leftIndex <= middle1 )
            combined[ combinedIndex++ ] = data[ leftIndex++ ];

      // copy values back into original array
      for ( int i = left; i <= right; i++ )
         data[ i ] = combined[ i ];

      // output merged array
      System.out.println( "         " + subarray( left, right ) );
      System.out.println();
   } // end method merge

   // method to output certain values in array
   public String subarray( int low, int high )
   {
      StringBuilder temporary = new StringBuilder();

      // output spaces for alignment
      for ( int i = 0; i < low; i++ )
         temporary.append( "   " );

      // output elements left in array
      for ( int i = low; i <= high; i++ )
         temporary.append( " " + data[ i ] );

      return temporary.toString();
   } // end method subarray

   // method to output values in array
   public String toString()
   {
      return subarray( 0, data.length - 1 );
   } // end method toString
} // end class MergeSort
```

Merge Sort Algorithm Test Program

```
// Figure 16.11: MergeSortTest.java
// Testing the merge sort class.

public class MergeSortTest
{
   public static void main( String[] args )
   {
      // create object to perform merge sort
      MergeSort sortArray = new MergeSort( 10 );

      // print unsorted array
      System.out.println( "Unsorted:" + sortArray + "\n" );

      sortArray.sort(); // sort array

      // print sorted array
      System.out.println( "Sorted:  " + sortArray );
   } // end main
} // end class MergeSortTest
```

Exercises

19.5 *(Bubble Sort)* Implement bubble sort—another simple yet inefficient sorting technique. It's called bubble sort or sinking sort because smaller values gradually "bubble" their way to the top of the array (i.e., toward the first element) like air bubbles rising in water, while the larger values sink to the bottom (end) of the array. The technique uses nested loops to make several passes through the array. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values are equal), the bubble sort leaves the values as they are. If a pair is in decreasing order, the bubble sort swaps their values in the array. The first pass compares the first two elements of the array and swaps their values if necessary. It then compares the second and third elements in the array. The end of this pass compares the last two elements in the array and swaps them if necessary. After one pass, the largest element will be in the last index. After two passes, the largest two elements will be in the last two indices. Explain why bubble sort is an $O(n^2)$ algorithm.

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);

}

}
```