Programming Assignment Divide and Conquer CS-254
Search and Sort

Goals: Advance skills in implementing pseudocode. Learn more about the Java ArrayList as the data structure used in the implementation. How to adjust an algorithm based on an array to the Java ArrayList. Understand and experience reduce or divide and conquer algorithms. Create tester code to show difference of O(n) to O(lg n) and O(n^2) to O(n lg n).

1. Create a new Java project in eclipse.
2. Add the provided SearchableAndSortable.java interface to the project.
3. Create a new class named SearchAndSortAlgorithms that implements this interface.
    a. You will implement each algorithm in this class.
    b. Adapt the algorithms to use an ArrayList as specified in the interface instead of an array.
4. Create a Driver class with a main method to test your algorithms. Be sure to test searching for the first and last elements, as well as for elements not in the list.

SearchAndSortAlgorithms
+ SearchAndSortAlgorithms(): //constructor empty
+ sequentialSearch(ArrayList<Integer> A, int K):int //provided Assignment 1
+ binarySearchRecursive(ArrayList<Integer> data, int value, int low, int high): int
+ binarySearchIterative(ArrayList<Integer> data, int value): int
+ insertionSort(ArrayList<Integer> A): void
+ quicksort(ArrayList<Integer> A): void
+ partition(ArrayList<Integer> A, int left, int right): int
+ printIntArrayList(ArrayList<Integer> data):void

Your code must implement the algorithms listed below. You must comment your code with exactly what part of the algorithm it is implementing. You must list Input, Output, and each line of the algorithm in comments by the code that implements it. No credit for code that does not have algorithm comments to match code.

- The *constructor does nothing. It is empty. You do not even need to make it as Java provides the default constructor if none specified.*
- sequentialSearch – already implemented previous assignment. Modify to use an ArrayList.

```
ALGORITHM Recursive BinarySearch(A[0..n-1], value, low, high)
//Searches for value in array A
//Input: A sorted array A[0..n-1] of n elements.
//Output: Index in array A of key or -1 if not found.
// initially called with low = 0, high = N-1
BinarySearch(A[0..N-1], value, low, high) {
      if (high < low)
          return -1
      mid = (low + high) / 2
      if (A[mid] > value)
          return BinarySearch(A, value, low, mid-1)
      else if (A[mid] < value)
          return BinarySearch(A, value, mid+1, high)
      else
          return mid
   }
```

```
ALGORITHM Iterative BinarySearch(A[0..n-1], value)
//Searches for value in array A
//Input: A sorted array A[0..n-1] of n elements.
//Output: Index in array A of key or -1 if not found.
BinarySearch(A[0..N-1], value) {
      low = 0
      high = N - 1
      while (low <= high) {
          mid = (low + high) / 2
          if (A[mid] > value)
              high = mid - 1
          else if (A[mid] < value)
              low = mid + 1
          else
              return mid
      }
      return -1
   }
```

```
ALGORITHM  InsertionSort(A[0..n − 1])
    //Sorts a given array by insertion sort
    //Input: An array A[0..n − 1] of n orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > v do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← v
```

Algorithm quicksort using Partition on next page:

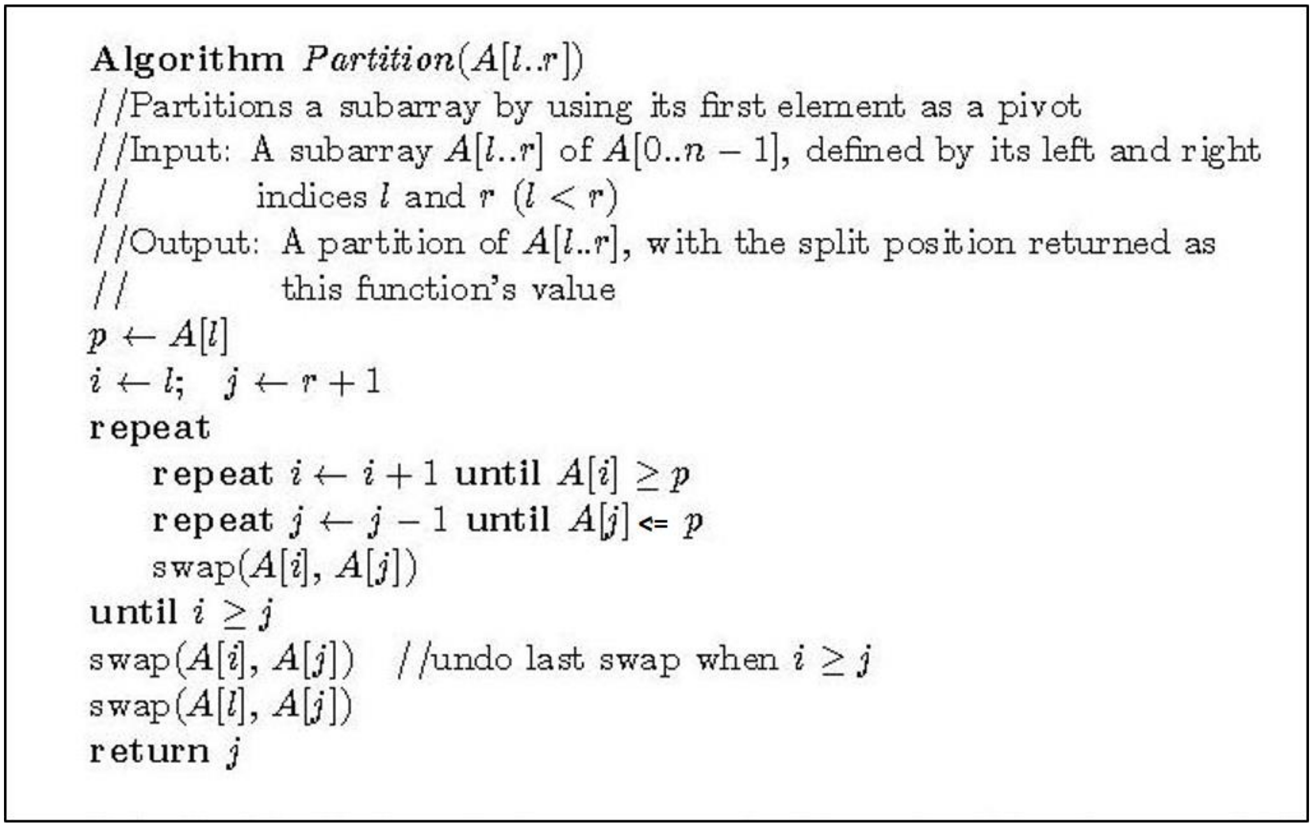
- quickSort – implement the algorithm in the method. (30 points)

```
/* Array arr[]
  low  <-- Starting index
  high <-- Ending index
*/
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Partition Algorithm:

Input: ArrayList<Integer> A, int left – left index, int right – right index of section of A to partition. (*left is l and right is r in algorithm*)



```
Algorithm Partition(A[l..r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//         this function's value
p ← A[l]
i ← l;   j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] <= p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```