Homework No. 6

Submit the all Java files developed to solve the problems listed below.

Problem No. 1

*(Quicksor*t) Consider now the recursive sorting technique called **Quicksort**. The basic algorithm for a one-dimensional array of values is as follows:

a) ***Partitioning Step****:* Take the first element of the unsorted array and determine its final location in the sorted array (i.e., **all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element**). We now have one element in its proper location and two unsorted subarrays.

b) ***Recursive Step****:* Perform step 1 on each unsorted subarray.

Each time step 1 is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, it must be sorted; therefore, that element is in its final location. The basic algorithm seems simple, but how do we determine the final position of the first element of each subarray? Consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

a) Starting from the rightmost element of the array, compare each element to **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 12, so **37** and 12 are swapped. The new array is

12 2 6 4 89 8 10 **37** 68 45

Element 12 is italicized+ bold to indicate that it was just swapped with **3**7.

b) Starting from the left of the array, but beginning with the element after 12, compare each element to **37** until an element greater than **37** is found, then swap **37** and that element. The first element greater than **37** is 89, so **37** and 89 are swapped. The new array is

12 2 6 4 **37** 8 10 ***89*** 68 45

c) Starting from the right, but beginning with the element before 89, compare each element to **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The new array is

12 2 6 4 ***10*** 8 **37** 89 68 45

d) Starting from the left, but beginning with the element after 10, compare each element to **37** until an element greater than **37** is found, then swap **37** and that element. There are no more elements greater than **3**7, so when we compare **37** to itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the above array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array. The algorithm has the following recursive structure:

```
procedure quicksort(arr: integer[], left, right: integer);
```

```
var i;
begin
if right > left then
begin
      i:=:partition(arr, left, right)
      quicksort (arr, left, i- 1) ;
      quicksort(arr,, i+l, right);
end
end
```

Using the preceding discussion, **<u>write recursive</u>** procedure **QuickSort** to sort a one-dimensional **Integer** array. The procedure should receive as arguments an **Integer** array, a starting index and an ending index. Method **partition()** should be called by method **quickSort()** to perform the partitioning step**.**

Write also a Java JApplet application with appropriate GUI to test the **Quick sort with an array of integer numbers of an arbitrary dimension (***allow the user to add the numbers from a JTextField to a JTextArea positioned on the left side of the applet by clicking a JButton, then use another JButton to quickSort the numbers displayed in the leftside JTextArea and display them in the rightside JTextArea***).**

<u>Задача 2</u>

Методът за сортиране **по метода на мехурчето**, показан на **Fig. 7.11 (**от ***лекция Ch-07Plus***, добавен за удобство в ***SampleCodeLab12b.rar*** **) не е ефективен за големи масиви**. Направете **следните промени** за да подобрите бързодействието на този алгоритъм.

a) След първия пас е сигурно, че най- голямото число се намера в края на масива, след втория пас двете най- големи числа са “на местата си” и т.н. Така вместо да се правят 9 сравнения на всеки пас за сортиране на 10 числа, променете този алгоритъм да прави 8 сравнения на втория пас, 7 сравнения на третия пас и т.н.
b) Напишете кода на програмата, така че изпълнението й да не зависи от дължината на масива, който се подава за сортиране
c) Ако масиваът е сортиран при задаването си, нужно ли е да се правят 9 паса а сортиране на 10 числа или по- малко паса ще свършат същата работа? Променете кода за сортиране да проверява в края на всеки пас дали са правени размени на елементи. Ако не са правени размени, то масива и вече нареден и програмата може да приключи. В противен случай се преминава към следващия пас.

<u>Задача 3</u>

(**Selection Sort**)

A selection sort searches an array looking for the smallest element in the array, then swaps that element with the first element of the array. The process is repeated for the sub array beginning with the second element. Each pass of the array places one element in its proper location. For an array of n- elements, n- 1 passes must be made, and for each sub array, n- 1 comparisons must be made to find the smallest value. When the sub array being processed contains one element, the array is sorted.

Write a **recursive** method **selectionSort** to perform this algorithm. **Write also a Java JApplet application** with appropriate GUI to test the **Selection sort with an array of integer numbers of an arbitrary dimension (***allow the user to add the numbers from a JTextField to a JTextArea positioned on the left side of the applet by clicking a JButton, then use another JButton to selectionSort the numbers displayed in the leftside JTextArea and display them in the rightside JTextArea***).**