Lab 06

Advanced Sorting Methods (Merge Sort, Shell Sort, Quick Sort)

Revised by Tran Thanh Tung

1. **Objectives**
   a. Know how, in reality, three advanced sorting methods work.
   b. Know how to use analysis tool to compare performance of advanced sorting algorithms to simple sorting algorithms (Bubble Sort, Selection Sort, Insertion Sort).
2. **Problem statement**
   a. Write a Java program to measure time (in seconds) needed for each simple sorting algorithms applying on ***the same random array*** of integer values. Sizes of arrays are accordingly 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000. Each time, you write down the measured time in following table.

	Merge Sort	Shell Sort	Quick Sort	Quick Sort Median-of-3
10000				
15000				
20000				
25000				
30000				
35000				
40000				
45000				
50000				

   b. Write some code to measure time (in seconds) needed for each simple sorting algorithms applying on ***Inversely sorted*** and ***Already-sorted order*** integer arrays of **50000** elements.

Table 1 - Experiment 2: Simple sorting in special cases

	Merge Sort	Shell Sort	Quick Sort	Quick Sort Median-of-3
Inverse order				
Already-sorted				

   c. Based on above table, give your comments on real complexity of the three simple sorting algorithms.

3. **Instruction:** (Follow instructions step-by-step)
   a. Take a look at sample source files (Lab_02) and read it carefully. There are three files: *i) Array.java*: contains class *Array*, *ii) TimeUtils.java*: contains class *TimeUtils* and *iii) SortingApp.java*: contains *main()* method.
   b. Add additional methods for Shell Sort and Quick Sort:

```
public void shellSort()
```

public void mergeSort()

public void quickSort()

public void quickSortMedOf3()

c. The code in *SortingApp* is now modified to measure time needed by *Bubble, Selection, Insertion, Shell, Quick* sorting methods.

```
int maxSize = 10000;          // array size
Array arr;                    // reference to array
arr = new Array(maxSize);  // create the array
arr.randomInit(maxSize);  // generate random array's elements
long startTime, endTime;      // get time just before running sorting
startTime = TimeUtils.now();
arr.xxxSort();            // corresponding sorting algorithm
endTime = TimeUtils.now();// get time just after running sorting
duration = endTime - startTime;      // time needed in milli-seconds
System.out.print("Time " + duration + "ms");
```

Note that: because after sorting by xxxSort, the array is changed. Thus you need to create the a copy of the original array to apply for each sorting method. Copy constructor *Array(Array oriArray)* can be used to create a copy of the array.

To check algorithms code correct or not, you can use *display()* method. It prints out content of an array.

d. Play around your program with different *maxSize* values and record their running time into a table as mentioned above. You can use MS Excel to store running times.

e. Ask your lab advisor how to produce a chart from your table.

f. Give some comments about running times of three simple sorting methods.

g. *Advance*: You notice that the running times depend on the random generated array. To eliminate it, you can run each sorting method several times (5, for example) and then take average. It's your task!

h. Finish.