CS526 Fall 2019
Homework 9

Due: 12/2

The goal of this assignment is to give students an opportunity to compare and observe how running times of sorting algorithms grow as the input size (which is the number of elements to be sorted) grows. Since it is not possible to measure an accurate running time of an algorithm, you will use an *elapsed time* as an approximation. How to calculate the elapsed time of an algorithm is described later.

You will use four sorting algorithms for this experiment: insertion-sort, merge-sort, quick-sort and heap-sort. A code of insertion-sort is in page 111 of our textbook. An array-based implementation of merge-sort is shown in pages 537 and 538 of our textbook. An array-based implementation of quick-sort is in page 553 of our textbook. You can use these codes, with some modification if needed, for this assignment. For heap-sort, our textbook does not have a code. You can implement it yourself or you may use any implementation you can find on the internet or any code written by someone else. If you use any material (pseudocode or implementation) that is not written by yourself, you must clearly show the source of the material in your report.

A high-level pseudocode is given below:

```
for n = 10,000, 20,000, . . ., 100,000 (for ten different sizes)
      Create an array of n random integers between 1 and 1,000,000
      Run insertionsort and calculate the elapsed time
      // make sure you use the initial, unsorted array
      Run mergesort and calculate the elapsed time
      // make sure you use the initial, unsorted array
      Run quicksort and calculate the elapsed time
      // make sure you use the initial, unsorted array
      Run heapsort and calculate the elapsed time
```

You can generate *n* random integers between 1 and 1,000,000 in the following way:

```
Random r = new Random( );
for i = 0 to n – 1
      a[i] = r.nextInt(1000000) + 1
```

You can also use the *Math.random*( ) method. Refer to a Java tutorial or reference manual on how to use this method.

Note that it is important that you use the initial, unsorted array for each sorting algorithm. So, you may want to keep the original array and use a copy when you run each sorting algorithm.

You can calculate the elapsed time of the execution of a sorting algorithm in the following way:

```
long startTime = System.currentTimeMillis();
call a sorting algorithm
```

```
long endTime = System.currentTimeMillis();
long elapsedTime = endTime - startTime;
```

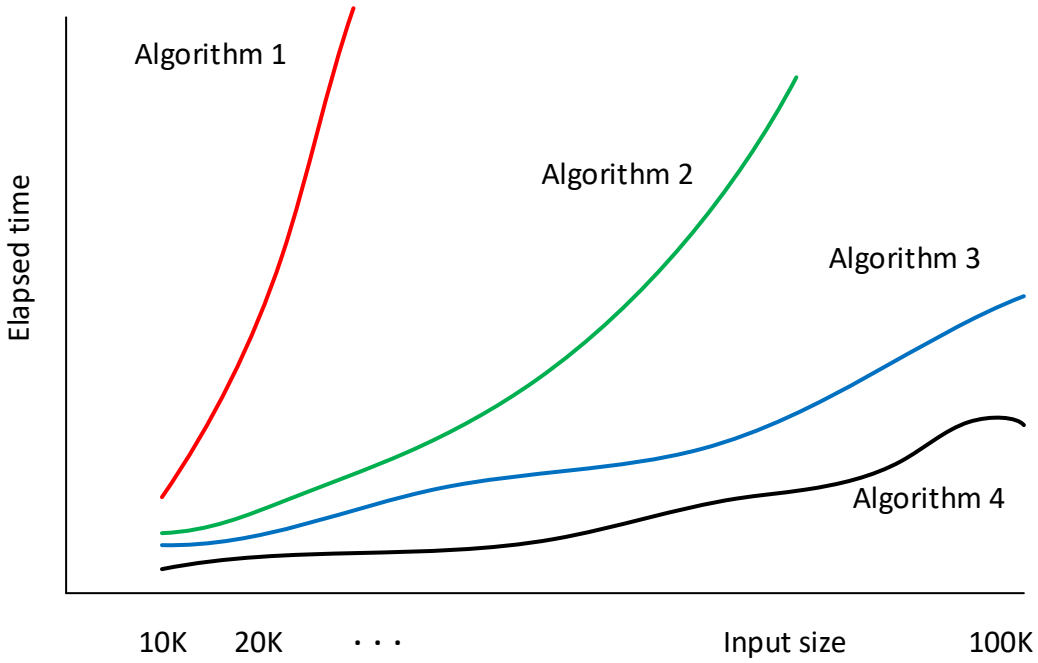
Collect all elapsed times and show the result (1) as a table and (2) as a line graph.

A table should look like:

n *Algorithm*	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
insertion										
merge										
quick										
heapsort										

Entries in the table are elapsed times in milliseconds.

A line graph should show the same information but as a graph with four lines, one for each sorting algorithm. The *x*-axis of the graph is the input size *n* and the *y*-axis of the graph is the elapsed time in milliseconds. Your graph should look like the following example (Note: this is just an example and your graph will look different):



You don't' need to write a Java program to generate the graph. Once you have all elapsed times, you can plot the graph using any other tools, such as a typical spreadsheet software.

Deliverables

You need to prepare two files. The first one is a document file that shows your experiment results. Name this file *hw9_doc.EXT*, where *EXT* is an appropriate file extension, such as *docx* or *pdf*. This documentation must include a table, a graph, and the discussion of the result along with what you learned from this experiment. The second file, named *SortingComparison.java* must have implementations of all four sorting algorithms and a main method. In the main method, you need to perform the experiment by invoking the four sorting algorithms. Combine the two files, and any other files that are needed by your program, into a single archive file and name it *LastName_FirstName_hw9.EXT*, where *EXT* is a file extension, such as *zip* or *rar*.

You need to include sufficient inline comments in your program.

Grading

If your results are not consistent with general properties of the sorting algorithms, up to 8 points will be deducted. Up to 4 points will be deducted if your discussion is not logically consistent and is not substantive. If there is not enough inline comments in your program, up to 4 points will be deducted.