Assignment 5

This assignment has 4 parts.

Part 1 (70 points). The goal of this part 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 below.

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;
```

Write a program that implements the above requirements and name it *SortingComparison.java*.

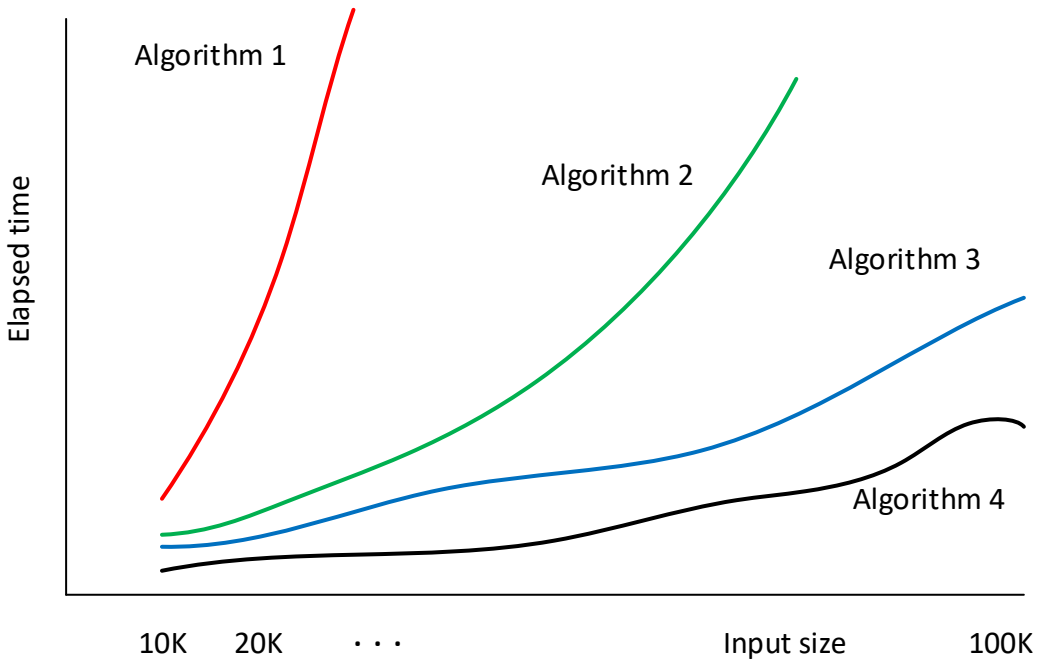
Collect all elapsed times and show the result (1) as a table and (2) as a line graph.

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.

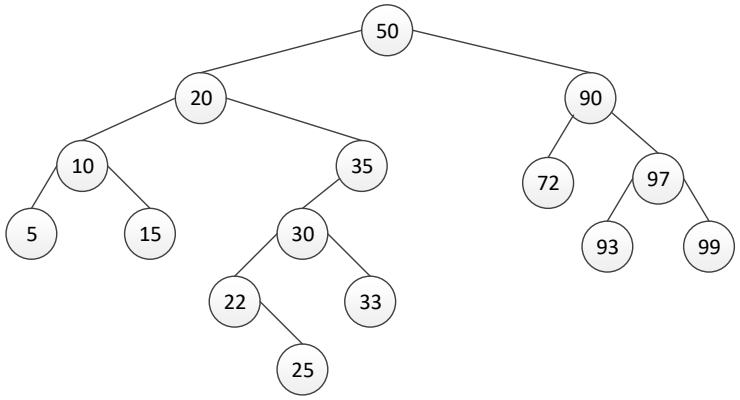
A line graph should show the same information but as a graph with four lines, one for each sorting algorithm. The *x*-axis of the graph is the input size *n* and the *y*-axis of the graph is the elapsed time in milliseconds. Your graph should look like the following example (Note: this is just an example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

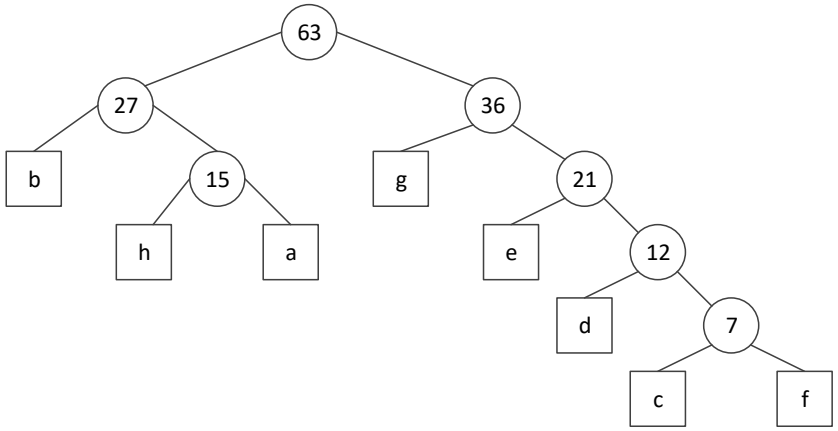
Note that, in your result, the running time of an algorithm may not increase as (theoretically) expected. It is possible that the running time of an algorithm may decrease a bit as the input size increases in a part of your graph. As long as the general trend of your graphs are acceptable, there will be no point deduction. So, you should not be too much concerned about that.

Part 2 (10 points). Consider the following binary search tree.



Suppose you delete the node with key = 50 (the root node). Show the resulting tree. You must use the method we discussed in the class (which is described in Chapter 11 note).

Part 3 (10 points). Consider the following Huffman code tree.



Part 3-(1). Encode a string "gdcha" using the above tree.
Part 3-(2). Decode 011110111100011111 to the original string.

Part 4 (10 points). In Chapter 13 note, a dynamic programming approach is illustrated using the "World Series Championship" example, and the following partially filled table is shown in the note:

$P(i, j)$

						1	6
						1	5
						1	4
					7/8	1	3
				1/2	3/4	1	2
			1/8	1/4	1/2	1	1
0	0	0	0	0	0		0
6	5	4	3	2	1	0	

j

i

Complete the table by calculating and entering values in all empty spaces, and show the completed table. You must show all calculations.

Deliverables

You need to submit the following files:

- *SortingComparison.java* and other files that are necessary to compile and run your program.
- *SortingComparison.pdf* file, which is a document as required by Part 1. This file must include:
  - A table and a graph that show your experiment result
  - Discussion of the result
  - What you learned from this experiment.
- *Hw5_others.pdf* file, which includes answers to Part 2, Part 3, and Part 4.

Combine the program file, additional files (if any), and the documentation file into a single archive file, such as a *zip* file or a *rar* file, and name it *LastName_FirstName_hw5.EXT*, where *EXT* is an appropriate file extension (such as *zip* or *rar*). Upload it to Blackboard.

Grading

- Part 1:
  - There is no one correct output. As far as your output is consistent with generally expected output, no point will be deducted. Otherwise, up to 30 points will be deducted.
  - If there are no sufficient inline comments, up to 10 points will be deducted.
  - If your conclusion/observation/discussion is not **substantive**, points will be deducted up to 10 points
- Part 2: Up to 6 points will be deducted if your answer is wrong.
- Part 3: Up to 3 points will be deducted for each wrong answer.
- Part 4: Up to 6 points will be deducted if your answer is wrong.