CS526
Homework Assignment 6

This assignment has two parts. Part 1 is an experiment that compares insertion times and search times of hash data structure, array list data structure, and linked list data structure. Part 2 is an experiment that compares the running times of four different sorting algorithms.

<u>Part 1 (50 points)</u>

The goal of Part 1 is to give students an opportunity to observe differences among three data structures in Java – HashMap, ArrayList, LinkedList – in terms of insertion time and search time.

Students are required to write a program named *InsertSearchTimeComparison.java* that implements the following pseudocode:

```
create a HshMap instance myMap
create an ArrayList instance myArrayList
create a LinkedList instance myLinkedList

Repeat the following 10 times and calculate average total insertion times and average
total search times for all three data structures

    generate 100,000 random integers in the range [1, 1,000,000] and store them in
    the array of integers keys[ ]

    // Insert keys one at a time but measure only the total time (not individual insert
    // time)
    // Use put method for HashMap
    // Use add method for ArrayList and Linked List

    insert all keys in keys[ ] into myMap and measure the total insert time
    insert all keys in keys[ ] into  myArrayList and measure the total insert time
    insert all keys in keys[ ] into myLinkedList and measure the total insert time

    // after insertion, keep the three data structures with all inserted keys.

    generate 100,000 random integers in the range [1, 2,000,000] and store them in
    the array keys[ ]

    // Search keys one at a time but measure only total time (not individual search
    // time)
    // Use containsKey method for HashMap
    // Use contains method for ArrayList and Linked List

    search myMap for all keys in keys[ ] and measure the total search time
    search myArrayList for all keys in keys[ ] and measure the total search time
```

search myLinkedList for all keys in keys[ ] and measure the total search time

Print your output on the screen using the following format:

```
Number of keys = 100000

HashMap average total insert time = xxxxx
ArrayList average total insert time = xxxxx
LinkedList average total insert time = xxxxx

HashMap average total search time = xxxxx
ArrayList average total search time = xxxxx
LinkedList average total search time = xxxxx
```

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

```
Random r = new Random(System.currentTimeMillis() );
for i = 0 to n - 1
    a[i] = r.nextInt(N) + 1
```

When you generate random numbers, it is a good practice to reset the seed. When you first create an instance of the Random class, you can pass a seed as an argument, as shown below:

```
Random r = new Random(System.currentTimeMillis());
```

You can pass any long integer as an argument. The above example uses the current time as a seed.

Later, when you want to generate another sequence of random numbers using the same Random instance, you can reset the seed as follows:

```
r.setSeed(System.currentTimeMillis());
```

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

We cannot accurately measure the execution time of a code segment. However, we can estimate it by measuring an elapsed time, as shown below:

```
long startTime, endTime, elapsedTime;
startTime = System.currentTimeMillis();
// code segment
endTime = System.currentTimeMillis();
elapsedTime = endTime - startTime;
```

We can use the *elapsedTime* as an estimate of the execution time of the code segment.

Part 2 (50 points)

The goal of part 2 is to give students an opportunity to compare and observe how running times of sorting algorithms grow as the input size grows. Since it is not possible to measure an accurate running time of an algorithm, you will use an *elapsed time* as an approximation as as described in the Part 1.

Write a program named *SortingComparison.java* that implements 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 input 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
```

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;
```

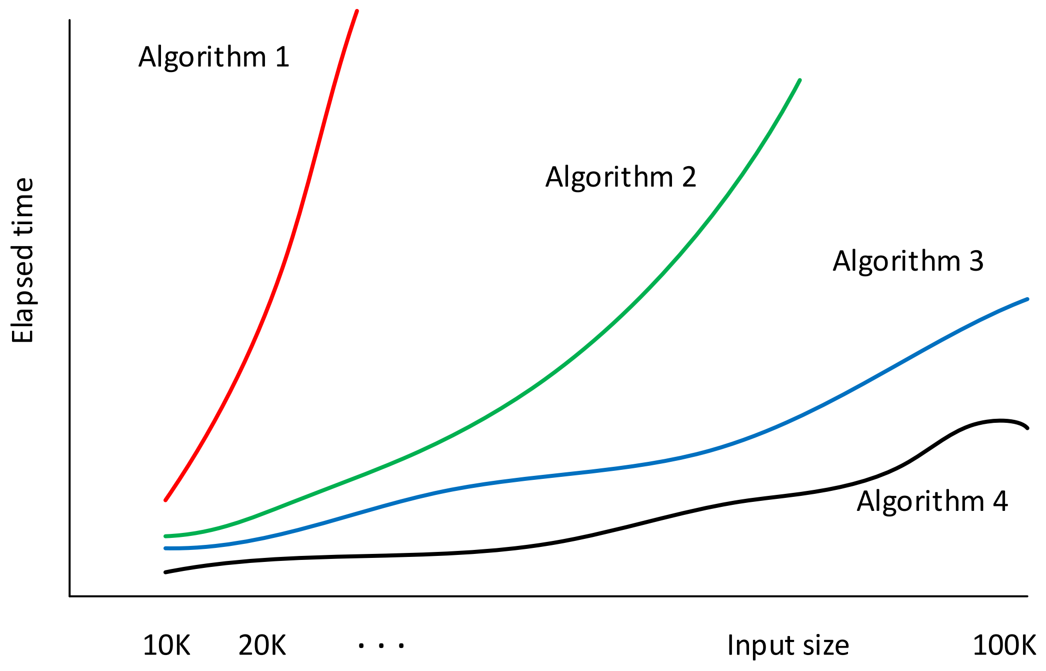
Collect all elapsed times and show the result (1) as a table and (2) as a line graph.

The 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.

The line graph shows 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. An example graph is shown below:



<u>Deliverables</u>

You need to submit program files and a documentation files.

Two program files to be submitted are *InsertSearchTimeComparison.java* and *SortingComparison.java* files. If you have other files that are necessary to compile and run the two programs, you must submit these additional files too

In a documentation file, you must include, for each part, your conclusion/observation/discussion of each experiment.

Combine all program files, 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_hw6.EXT*, where *EXT* is an appropriate file extension (such as *zip* or *rar*). Upload it to Blackboard.

<u>Grading</u>

For both parts, there is no one correct output. As far as your output is consistent with generally expected output, no point will be deducted. Otherwise, 10 points will be deducted for each part.

If your conclusion/observation/discussion is not **substantive**, points will be deducted up to 10 points.

If there is no sufficient inline comments, points will be deducted up to 20 points.