CSCI 2720
Assignment-4 Sorting Algorithms

For this project, you will implement Selection sort, Merge sort, Heap sort and Quicksort in JAVA and you will write an analysis report. **For quicksort, you should provide two implementations**. One implementation should use the first element of the array as the pivot. The second implementation should use a random pivot. All of your sorting algorithms need to keep track of the number of comparisons used by the sorting algorithm. **The sorting algorithms should sort a set of integer numbers in the ascending order.**

Start early on this assignment as you need to submit a report along with your code. Note that most of the code is already provided in the slide and you are free to use that code (just cite it in your read me file). The only thing you need to do is to calculate the number of comparisons each algorithm is making. For comparisons, you need to count when you compare two 'data elements' not the comparison for 'i' in a for loop (something similar to what was explained in the lecture).

Additional Requirements

1. You can do this assignment by writing your own classes with required data members and methods. You have full freedom on how you want to build your classes but we suggest creating a Sorting class with all the sorting algorithms as class methods and use an integer array to store the input and pass this integer array to these methods. **You don't need to use "ItemType" class or "Generics" data type in this assignment.**

2. As the program is run (with input filename as a command line argument as shown below), it should ask for which sorting algorithm to be used for sorting. Sorted results should then be generated using that algorithm for sorting.

3. Unlike the previous assignments, after printing the results of a sort, exit the program automatically. In other words, **do not** ask to enter the algorithm again in a loop.

4. In each of the sorting functions, you should embed statements to count the total number of comparisons used by the function for sorting. You can use a type long count variable to count comparisons. After sorting is completed, you should print this count exactly as shown in the example output.

5. For this assignment you need to do two sets of experiments and prepare a report which will include results, plots and discussion from these two experiments. The first experiment is explained in points 7 and 8 where you just need to find out the number of comparisons for

different sorting algorithms for different types of input files. The second experiment is explained in points 9 and 10 where you need to draw a plot between input size vs no. of comparisons for different sorting algorithms and then verify that the plots obtained by your experiments match with the theoretical results. You need to prepare a report summarizing all your results, plots, discussion and conclusion from these two experiments.

6. **You are given 3 input files for testing the first experiment. Check eLC contents page for these files.**
    - **a.** Ordered file - Containing integers placed in ascending order from 0 to 9999
    - **b.** Random file – Containing 10000 integers arranged in a random order
    - **c.** Reversed file – Containing integers placed in reverse order from 9999 to 0

7. You can use the same code snippet for reading input from the files as in the previous assignments. Read numbers from a text file and store it in an integer array and pass this array to your sorting functions. Write a well-organized report of your work. Your report must be submitted as a PDF file with your name in the middle of the cover page. Submit your report **(one report per group)** on elc and your code on Odin. Your report should include the following three points for experiment-1.

    a. Provide the total number of comparisons used by each one of the algorithms and explain whether they align with the Big O time complexity of each algorithm. In case of quick sort, compare and comment about the number of comparisons and complexity with the other quicksort implementation.

        i. For Ordered file as input.
        ii. For Random file as input.
        iii. For Reversed file as input.

        Please use the following table format to summarize the content for point a.

Algorithm	Input Type	# comparisons	Comments about time complexity and # comparisons
...	...	...	...

        After this table, also answer these questions below:

    b. Did you use extra memory space or other data structures other than the input array? If so, explain where and why?

    c. Explain what sorting algorithms work best in what situations based on your experimental results.

Here is a small example that shows what to expect when you run these algorithms on the input files. The size of input is 10,000 so for an algorithm with N^2 complexity N^2 is going to be 10000000 for such an algorithm. Expect 8-9 digits in the number of comparisons. Similarly for an algorithm with NlogN complexity NlogN will be 40,000. So expect 5-6 digits in the number of comparisons that you will

get. For an NlogN algorithm, if you are getting 8-9 digits in comparisons that will indicate a problem in your comparison calculation.

8. After completing the above program and report, you will do the next set of experiments where you will draw plots (n , input size vs no. of comparisons) for sorting algorithms and verify that the experimental results match with the theoretical results. You have complete freedom in how you want to implement this part of the assignment. You can create a new main, or add an function to your Sorting.java class or add an function in the SortDriver.java that implements this part of the assignment.Your code should use the algorithms created in Sorting.java to find additional insights about the 5 different sorting algorithms. For this part of the assignment you are not reading inputs from the text files, but instead you will be generating the inputs within the program itself and it should calculate the comparison values. Your code should be able to run the five algorithms (mentioned below) with different sizes of the inputs and give the number of comparisons. You can use these comparison values to draw the plots. (This is for the grading purpose) provide an interface in your implementation where user can specify which sorting algorithm they want to use and the size of the input that they wish to test. Your program should then generate an array with n number of **random values** (with n being the number specified by the user) and then it should sort those values using the sorting algorithm selected by the user. The number of comparisons should also be printed. **Please specify in your README file how to compile, run and use your implementation because it is mostly up to you for how you want to implement this part of the assignment. Also note there is no sample output for this part of the assignment.**

9. You are now going to use implementation for experiment-2 to create multiple different plots for the report mentioned above. Keep track of your experiments in a table formatted like the following and include this on your report:

Size of input

Algorithm	100	500	1000	5000	10000	20000	25000	30000
Selection								
Merge								
Heap								
QuickSort-fp								
QuickSort-rp								

Note: **I recommend running each algorithm multiple times and taking the average result for each input size to use in this table and for your plots. You can also use**

more values of n to make your plot look smoother. However, in the report you just need to show the results for the above table.

a. You should use some sort of graphing software like Microsoft Excel or Google Sheets to generate plots that compare the size of input n to the number of comparisons for the 5 sorting algorithms. You should have 5 separate plots(one for each sorting algorithm) with input size n on the x-axis and number of comparisons on the y-axis. Include these plots in your report.
b. You will then provide some discussion about your results. Compare the theoretical result with your experimental result for each algorithm. Does your experimental result coincide with the Big-O of that specific algorithm? Describe if there may be some inconsistencies between your plot and what the theoretical plot looks like.

Note:

1. **Your report must be in PDF format. Submit the report on the submission link provided on the eLC under the assignment section. Only one report per group is required.**
2. **Example output has not shown the complete list of numbers. However, you must print the complete list of numbers in your program. The number of comparisons shown in sample output below may be slightly different than the numbers that you get, however make sure that you at least get the same number of digits so you know that it is in the correct big-O.**
3. **quick-sort-fp stands for quick sort with first element as the pivot. quick-sort-rp stands for quick sort with random element as the pivot.**

Sample Output 1 (ordered.txt):

```
java SortDriver ordered.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: s
0 1 2 3 4 5 …….. 9999
#Selection-sort comparisons:  49995000

java SortDriver ordered.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: m
0 1 2 3 4 5 ……. 9999
#Merge-sort comparisons: 69008

java SortDriver ordered.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
```

```
Enter the algorithm: h
1 2 3 4 5 ……. 9999
#Heap-sort comparisons: 244576

java SortDriver ordered.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: q
1 2 3 4 5 ……. 9999
#Quick-sort-fp comparisons: 50004999

java SortDriver ordered.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: r
1 2 3 4 5 ……. 9999
#Quick-sort-rp comparisons: 290396
```

Sample Output 2 (random.txt):

```
java SortDriver random.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: s
1 2 3 4 5 ……. 9999
#Selection-sort comparisons: 49995000

java SortDriver random.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: m
1 2 3 4 5 ……. 9999
#Merge-sort comparisons: 120414

java SortDriver random.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: h
1 2 3 4 5 ……. 9999
#Heap-sort comparisons: 235440

java SortDriver random.txt
```

```
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: q
1 2 3 4 5 ……. 9999
#Quick-sort-fp comparisons: 303485

java SortDriver random.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: r
1 2 3 4 5 ……. 9999
#Quick-sort-rp comparisons: 299674
```

Sample Output 3 (reverse.txt):

```
java SortDriver reverse.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: s
1 2 3 4 5 ……. 9999
#Selection-sort comparison: 49995000

java SortDriver reverse.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: m
1 2 3 4 5 ……. 9999
#Merge-sort comparison: 64608

java SortDriver reverse.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: h
1 2 3 4 5 ……. 9999
#Heap-sort comparison: 226720

java SortDriver reverse.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: q
1 2 3 4 5 ……. 9999
#Quick-sort-fp comparison: 50009995
```

```
java SortDriver reverse.txt
selection-sort (s)  merge-sort (m)  heap-sort (h)  quick-sort-fp (q)
quick-sort-rp (r)
Enter the algorithm: r
1 2 3 4 5 ……. 9999
#Quick-sort-rp comparison: 292743
```

Note: Make sure to quit the program after printing the number of comparisons after each sort. In other words, do not ask to enter the algorithm again in a loop.

Grading Rubric

Grade Item	**Grade**
Selection Sort	10%
Merge Sort	10%
Heap Sort	10%
Quick Sort	10%
Report	50%
Readme, Comments & Specification conformity	10%
Total	**100%**

Your program should run with the following command syntax:

```
$ java <driver file> <input file name>
```

Commands to run and compile the code should be documented clearly in the Readme file.

The code that fails to compile or the code that compiles but fails to run will receive a grade of zero.

Late Submission Policy:

Except in the cases of serious illness or emergencies, projects must be submitted before the specified deadline in order to receive full credit. Projects submitted late will be subject to the following penalties:

- If submitted 0–24 hours after the deadline 20% will be deducted from the project score
- If submitted 24–48 hours after the deadline 40% points will be deducted from the project score.
- If submitted more than 48 hours after the deadline a score of 0 will be given for the project.
- **There is no assignment extension granted irrespective of the situation. If you need extra time then use the late penalty waiver option.**

Submission Notes:

You must **include your full name** and **university email address** in your **Readme.txt** file. If you are doing the project in a **group of two**, list the **full names and the email addresses of all two group members**. If you are in a group you must also **describe the contributions of each group member** in the assignment.

Contribution is expected to be 50% + 50%.

- **Report in PDF format (submit on eLC).**
- **Your java files**
- **3 input files**
- **Readme file with all necessary descriptions.**