COMP-10205 – Data Structure and Algorithms

Assignment #2
To Sort or not to Sort

Submission Requirements

Complete the following assignment and submit electronically in the assignments folder on myCanvas a single .java file. This assignment must be completed individually.

Background

You are to evaluate the sort algorithms that were presented in class for efficiency. You have been provided with starter code (see MyCanvas) that includes the source code for each of the sort methods discussed in the lecture (Bubble, Selection, Insertion, Merge, Quick and Radix sort). You must add code to each of the methods to count the number of comparisons required to completely sort the data, and compute the average time taken by each method. Ensure that you generate and use the same data for each sort. Remember that these sort methods replace the existing data (in place sorting), so you should keep a copy of your unsorted data. Use your student id as the random seed with the following code to generate a random array:

```
long myStudentID = 000123456; // replace with your student id
Random rand = new Random(myStudentID);
int[] array = new int[arraySize];
for(int i =0; i < array.length; i++) {
    array[i] = rand.nextInt(1,arraySize);
}
```

Use the supplied checksum method to validate data before and after each sort. Check that the data is randomized before the sort, validate the checksum after the sort. Each sorting method should produce the same checksum for a given data size, and report it.

Measure the time to sort the data using each method with **System.currentTimemillis**(). You will need to run many trials to get an average measurement. Use the following heuristic:

```
int totalRuns = (1000_000 / arraySize);
```

Requirements

Your program must report the following information:

1. Average sort time required in milliseconds (ms). Report all times up to 7 decimal places in accuracy. Report the number of operations and the time to execute a single operation (the comparison, or assignment in the case of Radix sort) in *ms* for each of the sort methods. The single operation time is determined by taking the time required to sort an array of a size *n* and dividing by the number of comparisons for that algorithm and then divide by the number of times the test was run. Report your results for data sets of 20, 400, 8_000 elements. Smaller array sizes will need to be averaged over a larger number of runs. Use the following code to generate a header for each test run:

```
int total_runs = (1000_000 / arraySize);
System.out.printf("\nComparison of sorts, Array size = %,d  total runs = %,d\n",
        arraySize, total_runs);
System.out.println("==============================================================");
System.out.println(
    "Algorithm      Run time     # of compares         ms / compares    checksum");
```

Use the following formatted print statements so your output is readable:

```
System.out.printf("%-10s", sortName);
System.out.printf("%,10.7f ms", execTimeMs);
System.out.printf("%,14d ops", compares);
System.out.printf("%,14.7f ms / op", msPerCompare);
System.out.printf("%,12d \n", checksum);
```

Columns *MUST* line up, and you must provide properly formatted output. Should look like this, only with ?? replaced by your actual test results.

```
Comparison of sorts, Array size = 400  total runs = 2,500
==============================================================
Algorithm      Run time     # of compares         ms / compares    checksum
aSort      0.00????? ms         ????? ops     0.0000??? ms / op     ???,???
bSort      0.00????? ms         ????? ops     0.0000??? ms / op     ???,???
cSort      0.00????? ms         ????? ops     0.0000??? ms / op     ???,???
dSort      0.00????? ms         ????? ops     0.0000??? ms / op     ???,???
eSort      0.00????? ms         ????? ops     0.0000??? ms / op     ???,???
fSort      0.00????? ms         ????? ops     0.0000??? ms / op     ???,???
```

You will need to produce a table for each data size. The results should be averaged across the total # of runs so your results are reliable.

2. Create a random array of 100,000 elements and then compare the performance of using a linear search (method a) vs a sort + binary search (method b) of the array. Search the array for -1 (it will not be found in the array). Use one of the 6 sorting techniques. Report the time required for both techniques and report the number of linear searches required that would justify sorting the data first. You will need to run the test repeatedly, and then take the average of multiple tests.

3. In a ***Comment* section at the TOP of your source file**, provide answers to the following:
   a. List sort algorithms in order of speed (fastest to slowest) when the array to be sorted contains 20 elements. Base this on your test results.
   b. List sort algorithms in order of speed (fastest to slowest) when the array to be sorted contains 8_000 elements. Base this on your test results.
   c. At 8_000 elements which sort has the lowest basic instruction set time? Does this impact your selection of the fastest algorithm? Why?
   d. In a table provide the following information for each algorithm:
      - Name
      - BIG O notation (time complexity – average case)
      - BIG O notation (space complexity – worst case)

   Example Output to include in comment

```
Sort Algorithm Identification
===========================================================
  Sort        Sort Algorithm      Big O      Big O
Algorithm          Name          (time)     (space)
------------------------------------------------------------
aSort             ________        O(___)     O(____)
bSort             ________        O(___)     O(____)
cSort             ________        O(___)     O(____)
dSort             ________        O(___)     O(____)
eSort             ________        O(___)     O(____)
fSort             ________        O(___)     O(____)
------------------------------------------------------------
```

Implementation Notes:

Part 1:

- You will need to make each method return the count of basic operations, or comparisons. Change the type of method from a void to a long method.

- You can use recursive counting for the recursive methods, or you may use a global variable to track each sorting algorithms performance, your choice. See the example binarySearchR() in the starter code for recursive counting.

- Your benchmarking program should be modularized so that code duplication is minimized. You should use method references (function pointers), and create a loop for each of the array sizes requested, and a loop for each test runs, and a loop for each sort type.

- You **MUST** include a statement of authorship at the top of your main method in a comment as follows: *I, <your name> <your student number>, certify that this work is my own work and that I did not consult external resources including artificial intelligence software to complete this assignment without due acknowledgement. I further certify that I did not provide my solution to any other students, nor will I provide it to future students taking this course at a later date.*

Marking Scheme - Rubric

Item	**Marks Available**
Well documented Code – use JavaDoc, fix ALL of the supplied code	2
Modular code – minimized duplicate code for counting	4
Part 1: Results Correct results for Part A:	4
Part 2: unsorted vs sorting searching – When should you sort?	4
Part 3: sort identification and performance analysis	6