Programming Assignment:
Empirical Study of Efficient Sorting Algorithms
CSC 310
Spring 2022

Introduction
In this programming assignment, you will implement and compare that actual run time of three popular and efficient sorting algorithms: **quick sort** (using a median-of-three pivot finding technique and a 3-way partitioning algorithm – see notes), **merge sort**, and **heap sort**. The purpose of this assignment is to give you experience implementing these algorithms for yourself and to conduct an empirical study the algorithm efficiency. You may complete this assignment in either Java, C, or python. **The programing assignment must be done independently.**

A Note on Coding Standard Algorithms
The point of the exercise is for you to code these classic algorithms yourself. It may be the last time you actually do this, because many programming languages include implementations of these algorithms in their APIs. Additionally, there are many available examples of code for these algorithms available on the internet.

You must write your own implementation. Do yourself a favor; avoid looking at code. If you do get stuck and decide to use code you find on the internet, it must be clearly commented. I must be able to see where the imported code starts and ends. **You will not receive credit for work that is not your own**, but you may use it in order to get some partial credit for the rest of the assignment.

If you do not properly credit any outside code that you use, I will treat that as a violation of academic integrity.

Directions
1. Write and test your implementation of each of the three algorithms. Keep in mind that
- sorted order is considered smallest to largest,
- for simplicity's sake, we will limit the values to integers between **[**0, 10,000**)**, that is 0 to 9,999
- eventually, you will probably want your programs to read input from a text file, and
- each algorithm should be able to handle repeated numbers.

2. Create 30 **randomly generated** files in each of the following categories

	unsorted	sorted (smallest to largest)	reverse sorted (largest to smallest)
small list (10,000 numbers)	30	30	30
medium list (100,000 numbers)	30	30	30
large list (1,000,000 numbers)	30	30	30

I recommend writing a separate program to create the 30 **unsorted** lists of each size and write them to **well-labeled** text files.

Notes:
- generating files this large requires time and memory. Here's a link on how to increase the heap space for a Java program that is launched from Eclipse. http://www.mkyong.com/eclipse/eclipse-java-lang-outofmemoryerror-java-heap-space/

- You may find that you need to increase the stack size on very large inputs. Before you do this to fix a StackOverflowError, make sure that everything runs correctly on the small inputs. Here is a link to change the stack size in Java from Eclipse: http://stackoverflow.com/questions/2127217/java-stack-overflow-error-how-to-increase-the-stack-size-in-eclipse

- In an attempt to pick a better pivot value, you should take the median of 3 values in the unsorted portion of the array. Use the first, last, and middle values in the unsorted portion of the array. Obviously, when the unsorted portion of the array gets very small the overhead is more costly than just picking the first value and pivoting around that. For this experiment, we shall say that *if the unsorted portion is less than 101 in length, we will simply use the first value in the unsorted portion of the array*.

- Since we are allowing repeated values, and since there will be many repeats in each list, we need to use a different partitioning algorithm than those discussed in the book. Wikipedia gives a decent explanation of the partitioning algorithm: https://en.wikipedia.org/wiki/Dutch_national_flag_problem

In this partitioning algorithm, we use "fat partitions". This means that we will partition the array into 3 sections: values *less than* the pivot, values *equal to* the pivot, and values *greater than* the pivot. The pivot value can then be placed at the beginning or end of the *equal to* section and be sorted.

Make sure that you only generate these files once, and not every single time you want to test something. You should expect that your input files will take about 500MB of disk space.

Once you have the unsorted files, you can either create to sorted and reverse sorted versions of the files by writing another program that uses a sorting algorithm provided in the API or you can create the sorted files manually using a tool like a spreadsheet program.

However you choose to create your dataset, *it is imperative that each algorithm be tested on the same dataset*. That way, there's no added confusion about the source of any performance difference you may detect.

3. Run your tests. However you organize it, your program should read in each test file and run each algorithm on that data. Your program should be in charge of recording the run time of each run in milliseconds and saving that information to an results file (text file). Do not do this process by hand, that is, do not run your test program once for each algorithm and input file.

4. Now import your results file into a spreadsheet program and calculate the average and standard deviation for each test case (unsorted small list, unsorted medium list, *etc.*).

5. Use the averages and standard deviations to write a 1-page report that summarizes your findings. Make sure you compare your findings to the theoretical times given by the number of basic operations each algorithm does and the growth order of each algorithm.

Deliverables

Please turn in the following in the dropbox on D2L:

- Your code. Make sure to include a readme file that explains how to compile and run your code.
  - If you write your code in C, you must include a makefile.
  - If you write your code in Java, please submit the Eclipse project AND each of the .java files that you wrote individually (not zipped)
- A zipped folder containing your data files. (This is really important! You should not upload 500M of files.)
  - Alternatively, you may give me a link to the folder where these files are stored in the could
- Your results file.
- Your 1-2 page report summarizing your findings.

Rubric:

1) Code is accompanied by clear user documentation. /5
2) Code compiles. /5
3) Code runs. /10
   (**note:** your code must run under the CS&E lab environment. If it does not run there, it doesn't run. IF your code does not run, I cannot give you grade for #5 and #6)
4) Data set provided. /10
5) Results file included. /10
6) Report is clear and explains the results well. /10

Total: /50