Project: Benchmarking Sorting Algorithms

Assignment weighting:
50% of the module grade

Due date:
See Moodle. **Late submissions will be penalised.**

Submission instructions:

Where code samples are requested, the code should be neatly laid out and formatted, and commented appropriately. Where diagrams are requested, computer-generated diagrams, as well as clear and legible scans or pictures of **neat** hand-drawn diagrams are acceptable. Where an explanation or discussion is requested, your answer is expected to be spell-checked, neatly laid out, and to use correct and appropriate grammar and terminology.

Your report may be written using a standard word processor (e.g. MS Word) or LaTeX. Please submit the final report in .pdf format. You should also include all source code which was written for the project. All the files for your project submission are to be uploaded to Moodle in a single .zip folder (**NOT** in a .rar, .tar.gz, .7z etc.), with the naming convention g00123456.zip, where g00123456 is your student number.

Failure to adhere to these instructions will incur a grading penalty.

Note on plagiarism and copying:

Plagiarism is passing off the work of another person as one's own.

While you are allowed to collaborate with your classmates and review online and print resources for high-level problem solving and background research, you are each expected to code, write and complete this assignment **individually**. If you use material from an external source (e.g. textbook, webpage, lecture notes) as part of your answer(s), you must explicitly acknowledge the source of the material.

Please see Section 4 of the GMIT Code of Student Conduct 2018/2019 for further information on plagiarism: https://www.gmit.ie/sites/default/files/public/general/docs/7-1-code-student-conduct-2018-2019.pdf

Plagiarism is a serious academic offence and may lead to a loss of marks and/or disciplinary proceedings.

Project Specification

For this project you will write a Python application which will be used to benchmark five different sorting algorithms. You will also write a report which introduces the algorithms you have chosen and discusses the results of the benchmarking process.

The five sorting algorithms which you will implement, benchmark and discuss in this project must be chosen according to the following criteria:

1. A simple comparison-based sort (Bubble Sort, Selection Sort or Insertion Sort)
2. An efficient comparison-based sort (Merge Sort, Quicksort or Heap Sort)
3. A non-comparison sort (Counting Sort, Bucket Sort or Radix Sort)
4. Any other sorting algorithm of your choice
5. Any other sorting algorithm of your choice

Python Application (40%):

This application should include implementations of the five sorting algorithms, along with a main method which tests each of them. Note that it is fine to reuse or adapt code for sorting algorithms from books, online resources or the lecture notes, as long as you add your own comments to the code and acknowledge the source.

To benchmark the algorithms, you should use arrays of randomly generated integers with different input sizes n. You should use a variety of different input sizes, e.g. n=10,n=100,n=500,…,n=10,000 etc. to test the effect of the input size on the running time of each algorithm. See the console output below for a selection of suggested sizes of n. You may test values of n which are higher than 10,000 if you wish, e.g. 500,000. Just be aware that algorithms such as Bubble Sort may take a long time to run when using large values of n!

The running time (in milliseconds) for each algorithm should be measured 10 times, and the average of the 10 runs for each algorithm and each input size n should be output to the console when the program finishes executing. See sample console output below (note that the output is formatted to 3 decimal places and laid out neatly):

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	0.102	0.149	0.403	0.480	0.793	1.180	4.242	11.394	21.123	37.690	47.000	63.786	95.543
Selection Sort	0.012	0.059	0.201	0.406	0.672	0.998	4.518	12.229	20.606	35.831	54.218	67.590	93.471
Insertion Sort	0.012	0.060	0.200	0.408	1.270	1.178	4.618	8.946	19.700	36.823	49.433	64.449	91.628
Counting Sort	0.011	0.031	0.056	0.070	0.018	0.021	0.025	0.023	0.030	0.037	0.062	0.031	0.032
Merge Sort	0.027	0.029	0.061	0.097	0.098	0.089	0.184	0.285	0.381	0.486	0.596	0.701	1.568

To measure the running time of a sorting algorithm, you should use Python's `time` module (https://docs.python.org/3/library/time.html) to record the start and end times in milliseconds, then subtract them to obtain the running time, as per the following code sample:

```
start_time = time.time()
# call your sorting algorithm here
end_time = time.time()
time_elapsed = end_time - start_time
```

The following code sample below may be useful – the function `random_array()` takes as input a value n and returns an array of n randomly generated integers with values between 0 and 99. You may use this code to generate random input instances which can be used when benchmarking your chosen sorting algorithms. Note you must import `randint` from Python's `random` module (https://docs.python.org/3/library/random.html) to use this code sample.

```
def random_array(n):
    array = []
    for i in range(0, n, 1):
        array.append(randint(0, 100))
    return array
```

Report (60%):

1. **Introduction (10%):** Introduce the concept of sorting and sorting algorithms, discuss the relevance of concepts such as complexity (time and space), performance, in-place sorting, stable sorting, comparator functions, comparison-based and non-comparison-based sorts, etc.
2. **Sorting Algorithms (5 x 5 = 25%):** Introduce each of your chosen algorithms in turn, discuss their space and time complexity, and explain how each algorithm works using your own diagrams and different example input instances.
3. **Implementation & Benchmarking (25%):** This section will describe the process followed when implementing the application above, and will present the results of your benchmarking. Discuss how the measured performance of the algorithms differed – were the results similar to what you would expect, given the time complexity of each chosen algorithm? In this section you should use both a table and a graph to summarise the results obtained (see samples below).

Sample results table – all values are in milliseconds, and are the average of 10 repeated runs

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	0.134	0.207	0.519	0.533	0.787	1.176	4.399	10.657	18.325	29.051	47.164	64.957	95.915
Selection Sort	0.012	0.059	0.202	0.406	0.674	1.005	3.754	9.002	26.442	34.325	52.65	75.001	94.126
Insertion Sort	0.015	0.078	0.262	0.539	0.802	1.184	4.851	9.004	17.727	29	44.216	68.007	94.588
Counting Sort	0.015	0.031	0.056	0.081	0.105	0.067	0.041	0.075	0.148	0.152	0.127	0.056	0.03
Merge Sort	0.042	0.032	0.062	0.098	0.132	0.169	0.239	0.291	0.386	0.491	0.515	0.593	0.676

Sample graph – note that the axes are labelled appropriately and include the correct units.