CSC3310 — Program #1 — Benchmarking Insertion and Selection Sorts

Objectives

- Write benchmarks
- Implement iterative sorting algorithms
- Apply asymptotic time complexity analysis to choose among competing algorithms

Overview

For this assignment, you are going to implement the insertion and selecting sort algorithms. You will then benchmark the run time of the two algorithms under the best, worst, and average case scenarios. You will plot the run times and interpret the plots in relation to the asymptotic run time.

Deliverables

Complete a lab report containing the following:

- Implementations of insertion sort and selection sort.
- Tests ensuring the two algorithms are implemented correctly.
- A benchmarking function
- Benchmark results for various input sizes *and* cases of both algorithms
- Analysis of the results
  - a regression model estimating asymptotic run time.
  - a series of plots comparing different input classes
- Answers to the the reflection questions

Each part is further explained below.

Submit your report to Canvas as a pdf.

N.B. You can use a Jupyter Notebook for your code and report, but are not required to do so. To do so, set up a notebook environment locally by installing the 64-bit Anaconda distribution (`https://www.anaconda.com/products/individual`). This writeup will include some tips for how steps may be accomplished in Python. All such steps can be accomplished through alternate means; using Python is not a requirement.

1 Instructions

1.1 Implement Insertion and Selection Sort

1. Create a Jupyter notebook for the project. It should have a title, your names, and an introduction.
2. Implement the following functions in the notebook:

   `void insertion_sort(lst)` takes a Python list and sorts it in place

   `void selection_sort(lst)` takes a Python list and sorts it in place.

3. Write tests (in the notebook) to ensure the two algorithms are implemented correctly.

1.2 Write a Benchmark Function

We will use benchmarking both to *validate* that the run time of your implementation agrees with the formal analysis and to *compare* the runtimes of two algorithms.

You will need to design a benchmark to measure the change in the run time of a sorting algorithm relative to changes in the size of the list of numbers being sorted.

`long benchmark(sorting_algorithm, input_list)` As input, the function takes both a sorting function and the list to sort. It returns the elapsed time in seconds.

You can use the following template:

```python
import time

# DO ANY SETUP

start_time = time.perf_counter()
# PUT CODE YOU ARE BENCHMARKING HERE
end_time = time.perf_counter()
elapsed = end_time - start_time
```

```java
// DO ANY SETUP

long start_time = System.nanoTime();
// PUT CODE YOU ARE BENCHMARKING HERE
long end_time = System.nanoTime();
long elapsed = end_time - start_time;
```

When designing your benchmark, keep the following in mind:

- Do not modify the input list object so it can be reused across benchmarks.

  For some sorting algorithms, the run time varies based on the order of elements (e.g., does better when a list is already sorted). After the first iteration, the list will be sorted and could throw off the benchmark results. You should make a separate copy of the original input list for each trial.

- Do not perform any data structures operations (e.g., list appends) inside the benchmark loop.

  For example, if you accidentally perform a $O(n)$ operation when trying to benchmark an $O(\log n)$ algorithm, the $O(n)$ operation will dominate and throw off your benchmark. If you need to do any setup, do it before the benchmark loop.

1.3 Design and Execute the Benchmarks

Every algorithm has three run-time cases: best, average, and worst. For our sorting algorithms, we will trigger these cases using sorted, randomly shuffled, and reverse-sorted input lists, respectively. You will need to perform 6 benchmarks for each list size (2 algorithms $\times$ 3 cases).

You will need to choose the list sizes. A few things to keep in mind:

- The dominating term in the run time complexities only dominate for large enough input sizes.
- Run times can vary due to garbage collection, OS scheduling, etc. The magnitudes of your measured run times must be larger than the magnitude of the noise. You can control this by using larger list sizes. You may also want multiple runs.
- List sizes should vary be orders of magnitude (e.g., 100, 1000, etc.).
- You should benchmark at least 5 (large!) list sizes to be able to reliably differentiate between linear and non-linear behavior.
- 100 is not a large list size.

1.4 Validating Formal Run Times

We can estimate the run time complexity function from the measured run times using a little bit of statistics. If we fit a linear regression model to the logarithms (base doesn't matter) of the list sizes (s) and run times (r) to estimate the slope (m):

$$\log r = m \log s + b$$

Then the slope tells us the exponent of the growth function:

m	Run Time
0	Constant
< 1	Sub-linear ($\log n$, $\log \log n$, $\sqrt{n}$, ...)
1	Linear
$1 < m < 2$	Between linear and quadratic ($n \log n$, $n\sqrt{n}$, ...)
2	Quadratic (n^2)
$2 < m < 3$	Between linear and quadratic ($n^2 \log n$, ...)
3	Cubic (n^3)

You can calculate the slope (m) using the following code snippet:

```
import numpy as np
from scipy.stats import linregress
m, b, _, _, _ = linregress(np.log(list_sizes), np.log(run_times))
```

If you are not using Python,you can use a spreadsheet or other tool too perform the regression.

It is unlikely that your data will be perfect; as such you shoud not treat the above table rigidly.

1.5 Comparison of Algorithm Run Times

We want to make the following comparisons:

- For each algorithm, compare the run times of the three cases
- For each case, compare the run times of the two algorithms

To do this, you will make a series of plots of the benchmark data in different combinations. For example, to compare the run times of multiple cases for a single algorithm, you would create a plot with 3 lines (one for each case). The horizontal axis would give the list sizes, while the vertical axis would give the run times. In total, you will create 5 plots (1 for each algorithm with the 3 cases, 1 for each case with the 2 algorithms)

You can use `matplotlib` to create the plots. Add a title, label each line, label the axes, and create a legend.

```
import matplotlib.pyplot as plt
plt.plot(list_sizes, run_times_best, label="best ")
plt.plot(list_sizes, run_times_average, label="average")
plt.plot(list_sizes, run_times_worst, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Insertion Sort", fontsize=20)
plt.legend()
```

1.6 Reflection Questions

- Create a table of the theoretical and estimated run time functions for the 6 combinations (2 algorithms, 3 cases). Do your estimates match the theory? If not, you may have made a mistake somewhere.
- Which algorithm had a better run time than the other and for which case(s)? Why do you think that one case was substantially faster for that algorithm? (Hint: focus on the inner loops.)
- Based on your results, which of the two sorting algorithms would you use in practice? Why?

1.7 Appendix

In addition to including your sort implementations in the report, include a zip archive containing your full source code as an "appendix".

Rubric

I will be looking for:

- Name, a title, and an introduction, including your own summary of the lab.
- Code is to a high technical quality, with little-to-no duplicated code and clear control flow.
- Benchmark data is useful.
- Benchmarks and analyses are accurate
- Plots are resonable, clear, and labelled.

Followed submission instructions	5%
Writing and Presentation	10%
Algorithms: implementations and tests	20%
Benchmarks: input data, output data, execution	30%
Analysis: linear regression and plots	20%
Reflection questions	15%