Lab 1: Benchmarking Insertion and Selection Sort

CS 3851 Algorithms

Learning Outcomes

- Learn how to write benchmarks in Python
- Implement iterative sorting algorithms such as insertion sort and selection sort
- Apply asymptotic time complexity analysis to choose among competing algorithms

Overview

You may use either a Jupyter Notebook running on your laptop or ROSIE. Nothing we will do will require GPUs or more compute resources than your laptop has. Should you choose to work on your laptop, the easiest way to setup a Python Notebook environment is with the 64-bit Anaconda Distribution (https://www.anaconda.com/products/individual).

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.

Instructions

1. Implement Insertion and Selection Sort

1. Create a Jupyter notebook named lastname_lab01. The notebook should have a title, your name, and an introduction.

2. Implement the following functions in the notebook:

- `void insertion_sort(lst)` – The function takes a Python list and sorts it in place.
- `void selection_sort(lst)` – The function takes a Python list and sorts it in place.

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

2. Write a Benchmark Function

Benchmarking can be used to (1) validate that the run time of an algorithm implementation agrees with the formal analysis and (2) compare the run times of two or more algorithms.

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

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

You can use the following template:

```
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
```

When designing your benchmark, keep the following in mind:

1. 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.

2. 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.

3. Design and Execute the Benchmarks

Every algorithm has three run-time cases: best, average, and worst. Multiple or all of the cases may have the same run time. For sorting algorithms, the cases are triggered by sorted, randomly shuffled, and reverse sorted input lists. You will need to perform 2 algorithms x 3 cases = 6 benchmarks for each list size. You will need to choose the list sizes. A few things to keep in mind:

1. The dominating term in the run time complexities only dominates for large enough input sizes.

2. 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.

3. List sizes should vary by orders of magnitude (e.g., 100, 1000, etc.).

4. You should benchmark at least 5 list sizes to be able to reliable differentiate between linear and non-linear behavior.

4. Validating Formal Run Times

We can estimate the run time complexity function from the measured run times using a little bit of statistics.

1. 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):

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

2. The slope tells us the exponent of the growth function:

m	Run Time
0	Constant
< 1	Sub-linear (e.g., log n)
1	Linear
1 < m < 2	Between linear and quadratic (e.g., n log n)
2	Quadratic (e.g., n^2)
2 < m < 3	Between linear and quadratic (e.g., n^2 log n)
3	Cubic (e.g., 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))
```

5. Comparative Analysis of Algorithm Run Times

We want to make the following comparisons:

1. Compare run times of the three cases within each algorithm

2. Compare run times of each case across all of the algorithms

You can do this by making 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 have the list sizes, while the vertical axis would have 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. You may need to take the $\log_{10}$ of the list sizes to get a nice graph.

```
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()
```

6. Reflection Questions

1. 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.)

2. Which algorithm had a better run time than the other and for which case? Why do you think that one case was substantially faster for that algorithm? (Hint: focus on the inner loops.)

3. Based on your results, which of the two sorting algorithms would you use in practice?

Submission Instructions

Save the notebook as a PDF and upload it to Canvas.

Rubric

I will be looking for the following:

- A name, title, and introduction (including your own summary of the lab) at the top of the notebook. Make sure to put your name at the top of the notebook.
- Code is to a high technical quality with little or no duplicated code
- That your benchmarks and analyses are correct
- That your line plots look reasonable

Followed submission instructions	5%
Overall quality of the presentation of the notebook including plot labels	10%
Algorithm implementations	10%
Algorithm tests	5%
Benchmark function is correct	15%
6 benchmarks executed correctly	15%
Correct estimation of growth function orders	10%
5 comparisons plots are correct	10%
Plots of run times are correct	10%
Answers to reflection questions	10%