ECE 590 Project 1 Report

Name: Shu Yu Lee NetID: sl603
Name: Fang Feng NetID: ff34

Our program, for both unsorted and the sorted input arrays, behave as expected. Selection sort, insertion sort and bubble sort run in approximately O(n^2) time, while quick sort and merge sort run in O(nlogn) time. Our quick sort algorithm behaves better when we have unsorted input, because we decided to use the first element as our pivot value during partition. When the input is sorted, our quick sort algorithm will divide the array to 1 and n-1 which is the worst case for quick sort leading to worse time complexity results.

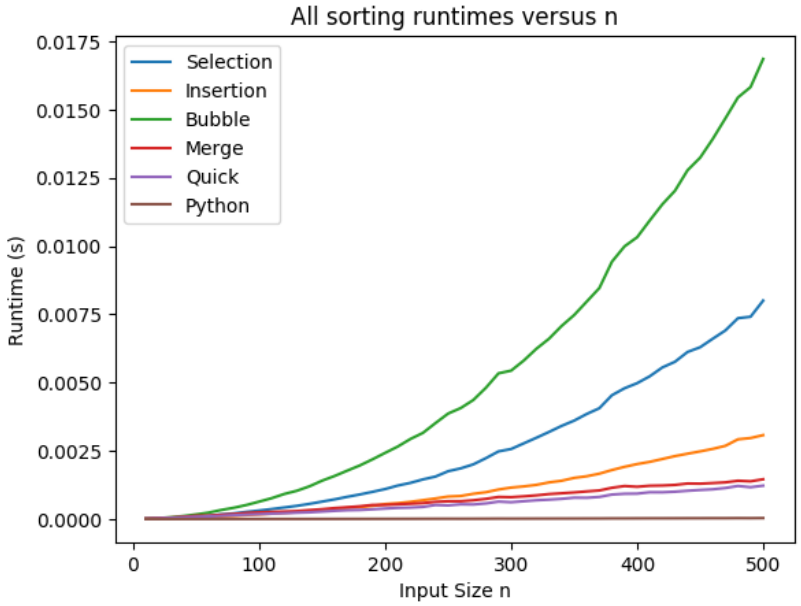
```
Selection Sort log-log Slope (n>400): 2.033682
Insertion Sort log-log Slope (n>400): 2.097396
Bubble Sort log-log Slope (n>400): 2.132665
Merge Sort log-log Slope (n>400): 1.100274
Quick Sort log-log Slope (n>400): 2.005688
```

Sorted array

```
Selection Sort log-log Slope (n>200): 2.192515
Insertion Sort log-log Slope (n>200): 1.915111
Bubble Sort log-log Slope (n>200): 2.127818
Merge Sort log-log Slope (n>200): 1.184646
Quick Sort log-log Slope (n>200): 1.279541
```

Unsorted array

According to the Big-O time complexity analysis we learnt from the class, quick sort and merge sort belong to the class O(nlogn), and the rest of the algorithm is in class O(n^2) which is much slower than the class O(nlogn). And our result is consistent with this expectation. Based on our experiment results, quick sort is the best algorithm and the babble sort is the worst. The possible reasons we conclude is that merge sort needs to copy the original array to a new space. which is relatively time consuming, while quick sort only allocate space on stack for recursive call. Therefore, quick sort is better than merge sort. On the other hand, among the three O(nlogn) algorithms, babble sort requires the most swapping during execution, which lead to longer runtime comparing to the rest of the algorithms.



In our runtime analysis, we decided to focus on the performance of algorithm on large n with multiple trials. Reporting theoretical runtimes for asymptotically large values of n is to

be more consistent with the assumption we made in Big-O time complexity analysis that the leading term of n is the dominating term, regardless of any constant factors. When n is asymptotically large values, the constant factor of the algorithms won't have that much effect on the comparison between the time complexity among algorithms. When we ran with a smaller value of n, the results are more biased with the correct time complexity. Because when n is smaller, the affection on the runtime of other elements will be more obvious. The constant factor in our runtime may become the dominating term.

Also, the runtime of a single trial may be influenced by many different factors. For example, if an urgent task (something with high priority) comes in during the execution of our sorting program, our program, depending on the scheduling scheme of different operating system, may be halted, then resume execution after that important task is completed. In this situations, when we compute the runtime of our algorithm, we actually include not only the runtime of our sorting, but also the time of executing that task. If we assume that this kind of tasks are not frequent, the mean of multiple trials returns a better estimation of our program runtime.

Due to what we mentioned about CPU scheduling, theoretically, we are supposed to see worse performance in our program runtime if we run the program with some computationally expensive tasks. However, there is not any significant difference in the runtime when we open some internet browsers during our program execution. The most possible reason might be that we did not actually exhaust out CPU. We ran our program on a MacBook Pro with with a 6-core Intel i7 processor. And our program does not utilize concurrency. We perform our sorting algorithms and all of testing in serial, which means that the runtime would not be affected whenever there is at least one idle processor core.

Then we try to stress our CPU by executing multiple “yes” command, which basically consume all of the processor resource. Before we max out our CPU, the testing result of our sorting algorithms is:

Selection Sort log-log Slope (n>200): 2.113915
Insertion Sort log-log Slope (n>200): 1.837432
Bubble Sort log-log Slope (n>200): 2.045496
Merge Sort log-log Slope (n>200): 1.081577
Quick Sort log-log Slope (n>200): 1.182780

After maxing out our CPU, the result becomes:

Selection Sort log-log Slope (n>200): 2.262761
Insertion Sort log-log Slope (n>200): 1.941845
Bubble Sort log-log Slope (n>200): 2.146480
Merge Sort log-log Slope (n>200): 1.235926
Quick Sort log-log Slope (n>200): 1.333706

We can see some increase in the runtime of all of our algorithm. This result implies that when it come to experiment, there are many factors which we are not able to control can affect program runtime. The actual runtime of a program many be different under different situation, since CPU scheduling scheme, hardware architecture, or even the implementation of that algorithm can influent the runtime. Theoretical runtime provides us a rather mathematical standard to compare the performance of different algorithms. If we want to predict the performance of an algorithm, regardless of input data and machine of execution, theoretical runtime is a more suitable choice. On the other hand, if we want to test the performance of a particular program, on a particular data set, or a particular machine, we will need to actually execute the program for a more accurate runtime.