The runtimes of these algorithms seem to behave largely as expected (see Figure 1), acknowledging that there is some experimental error. Looking at the log-log slopes for unsorted data with n > 200, we see that selection sort, insertion sort, and bubble sort are running close to $O(n^2)$, which approximates their theoretical runtimes in the average case. Merge sort and quick sort have slopes between 1 and 2, which also corresponds to their O(nlogn) theoretical runtimes in the average case. While the slope for quick sort is larger than the slope for merge sort, I am cautious to draw any conclusions about the relative speeds of these two algorithms from log-log slopes alone because O(nlogn) runtimes do not produce straight lines on log-log plots. Looking at the log-log slopes for sorted data with n > 400, we again see agreement with theoretical run-times in the best case. Selection sort runs $O(n^2)$, while insertion sort and bubble sort run O(n), and merge sort and quick sort run have slopes between 1 and 2.

Figure 1: Testing code output

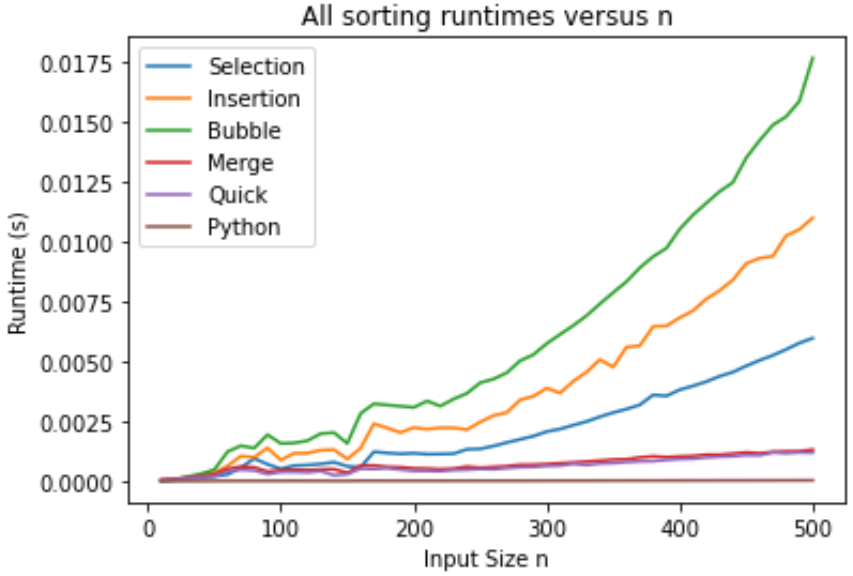
```
UNSORTED measureTime

Timing algorithms using random data.
Averaging over 30 Trials


Selection Sort log-log Slope (all n): 1.859979
Insertion Sort log-log Slope (all n): 1.932436
Bubble Sort log-log Slope (all n): 1.985477

Selection Sort log-log Slope (n>200): 2.080117
Insertion Sort log-log Slope (n>200): 2.034196
Bubble Sort log-log Slope (n>200): 2.034790
Merge Sort log-log Slope (n>200): 1.122637
Quick Sort log-log Slope (n>200): 1.269071

SORTED measureTime

Timing algorithms using only sorted data.


Selection Sort log-log Slope (all n): 1.977019
Insertion Sort log-log Slope (all n): 1.028097
Bubble Sort log-log Slope (all n): 1.029064

Selection Sort log-log Slope (n>400): 2.037440
Insertion Sort log-log Slope (n>400): 1.003043
Bubble Sort log-log Slope (n>400): 1.005805
Merge Sort log-log Slope (n>400): 1.157084
Quick Sort log-log Slope (n>400): 1.199556
```

This project has demonstrated the utility of analyzing both theoretical runtimes and performing experiments. By understanding theoretical runtimes for these algorithms in

different scenarios beforehand, I was better able to diagnose whether my implementations were correct. For example, my insertion sort originally had a bug that caused it to have $O(n^2)$ runtime on sorted data, which I knew was not correct due to the derived theoretical runtime. Aside from aiding in proper implementation of algorithms, theoretical runtimes allow for comparisons of runtimes without (1) having to actually code up the algorithms and (2) being dependent on the performance of different machines. That said, I imagine that experimental runtimes are more useful for some real-life applications (e.g., a quant trading firm cares more about how quickly it can run an algorithm and get a message to an exchange than it does about a theoretical runtime).

We report theoretical runtimes for asymptotically large values of n because we are interested in algorithms' performance as the amount of data becomes large. For small amounts of data, the practical difference in algorithm performance (among a reasonable suite of algorithms) is not likely to be very significant. When experimentally looking at smaller values of n, the log-log slopes differ slightly but not with any intuitive trend. Additionally, I note that the runtimes for all of the algorithms look quite similar, likely due to the fact that the difference between $O(n)$ and $O(n^2)$ complexity is relatively minor for small n compared to fixed computing costs and background noise. When timing algorithms, we average runtimes across multiple trials because runtimes can be quite variable from one trial to the next, but a clear trend of runtime vs. input size is seen when we average out that variability over a moderately large number of trials. When we use only a single trial, we see some unreasonably "wiggly" behavior in the graph of runtime vs. input size (see Figure 2). All runtimes in this report were calculated when I had many internet browser windows opened (i.e., while performing computationally expensive tasks in the background). When I close all of these windows and run the timing code again, I observe a very small reduction in runtime.

Figure 2: Unsorted runtimes with numTrials = 1



I would say that the "best" and "worst" sorting algorithms depend on whether we're looking at sorted data or unsorted data (see Figures 3 and 4). In the unsorted case, merge sort

and quick sort performed similarly well (best, not including the Python implementation), and our experimental results are likely not reliable enough to distinguish the two. In the sorted case, bubble sort performed best (not including the Python implementation), and selection sort performed worst. If I had to choose a "best" and "worst" overall algorithm, I would choose selection sort as the "worst" because it runs $O(n^2)$ even in the best case, and I'd choose quick sort as the best because it performs relatively well in both the sorted and unsorted cases and performs sorting in place.

Figure 3: Runtimes on unsorted data

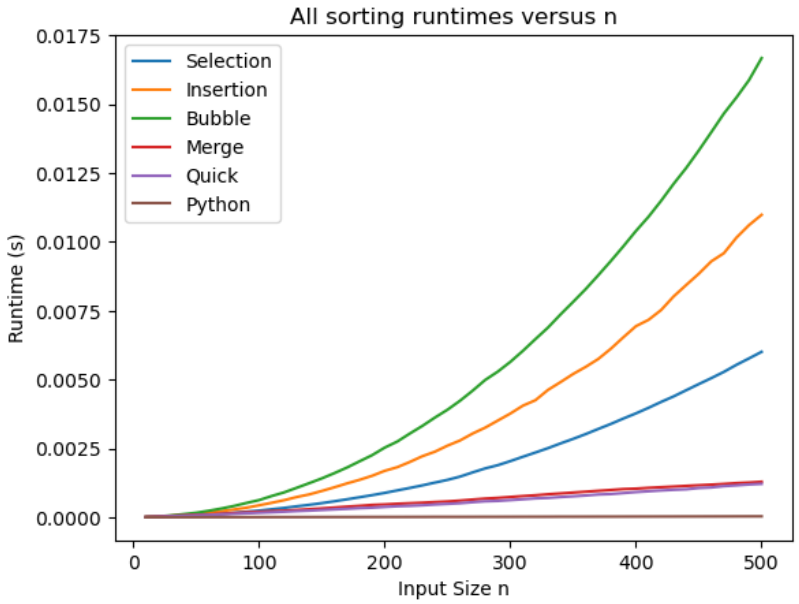


Figure 4: Runtimes on sorted data