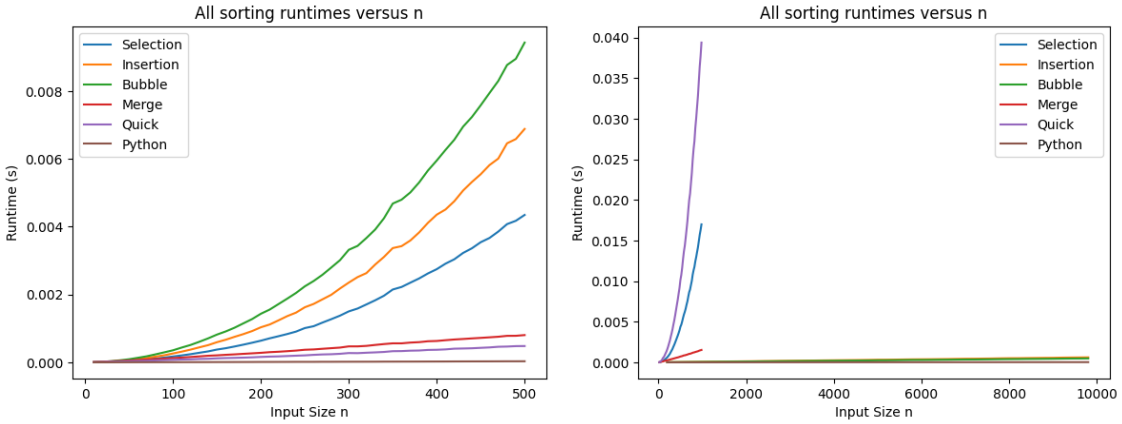
ECE 590 - Project 1

Koushik Annareddy Sreenath (ka266) Shravan Mysore Seetharam (sm952)

1) Do your algorithms behave as expected for both unsorted and sorted input arrays?

1. Bubble sort: It performs better if the elements are sorted. As iteration stops when there are no swaps made.
2. Insertion sort: it performs better when the input is sorted. This is because it will not iterate in the inner loop.
3. Selection sort : it performs similarly for both sorted and unsorted lists. Since it always searches for the minimum element in the sublist.
4. Merge sort: It performs similarly for both sorted and unsorted lists. Because it always divides the list into two halves recursively no matter the type of input.
5. Quick sort: It performs worse when the list is sorted and pivot is the last element (our implementation). Since for each iteration the sub list size would be n-1. Therefore, the time complexity increases to O(n^2).



2) Which sorting algorithm was the best (in your opinion)? Which was the worst? Why do you think that is?

Based on our observation we prefer Merge Sort to be a better sorting algorithm for the following reasons.

1. No matter what the input is (i.e. sorted list or unsorted list). Merge sort maintains uniform time complexity of O(n*log(n)). Even though quick sort might perform better if we pick the right pivot point. However it has higher time complexity O(n^2) if the pivot is not chosen appropriately.
2. Even though the Space complexity of Mergesort is more than other sorting algorithms, we believe that memory cost is significantly cheaper than performance cost.

Based on our observation we believe that selection sort is the worst performing sorting algorithm for the following reasons.

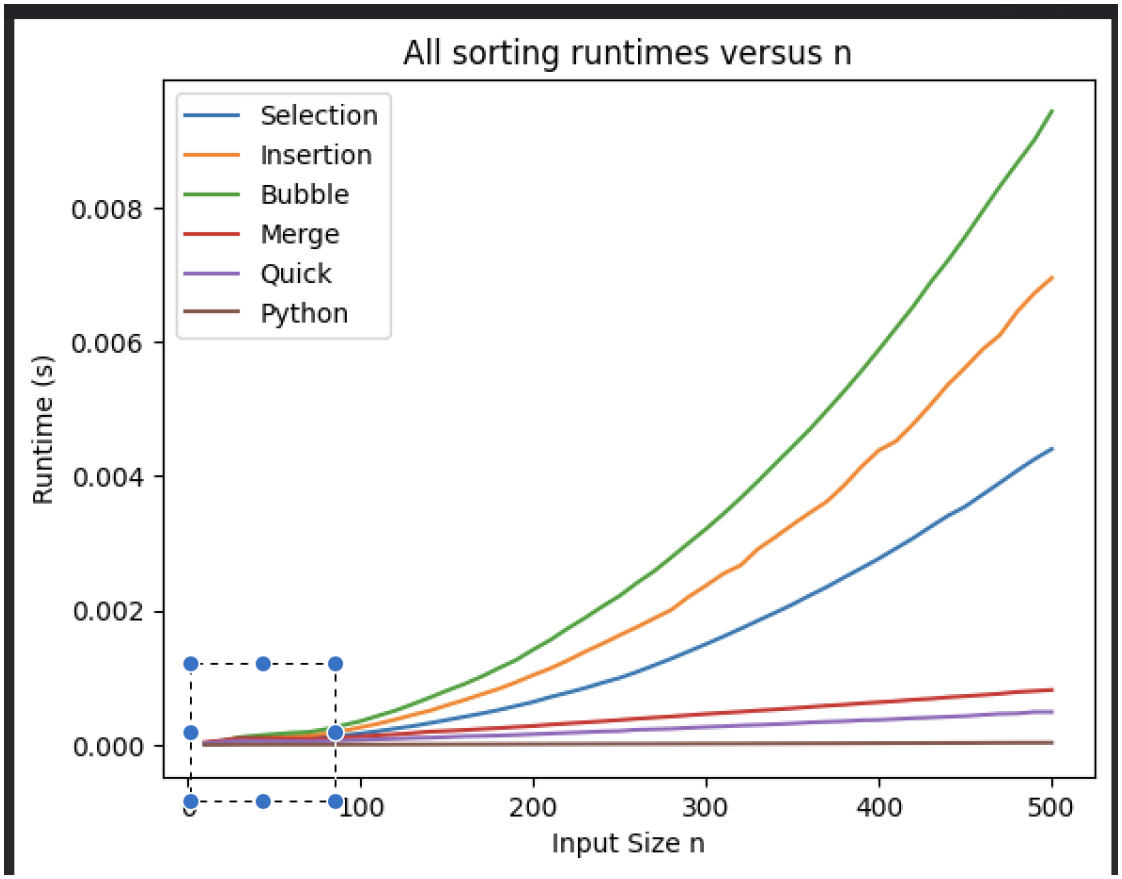
1. No matter the input, selection sort always iterates n^2 times. Other sorting algorithms perform better for some inputs. However, selection sort is always O(n^2)

3) Why do we report theoretical runtimes for asymptotically large values of n?

We cannot always compute exact runtime cost for asymptotically large values of n. Hence, we generalize the Time and Space complexities using theoretical runtime. This simplifies assessing the performance costs of different algorithms.

4) What happens to the runtime for smaller values of n? Why do you think this is?

For small values of n (i.e less than 50) all the above sorting algorithms will perform comparatively similar.



This is because for lower values of n the time cost difference between a better performing algorithm and a lower performing algorithm will be very less. Hence it would not matter that much for small values of n.

5) Why do we average the runtime across multiple trials? What happens if you use only one trial?

Certain algorithms might perform good / bad for certain inputs. Therefore IF we are doing only one trail, it would not be a fair representative of how an algorithm performs for varied input. Hence we should always compute average runtime across multiple trails to decrease the influence of input biases.

6) What happens if you time your code while performing a computationally expensive task in the background (i.e., opening an internet browser during execution)?

We would see a proportional growth of the runtime of the algorithms based on its time complexities when a computationally intensive task is running in the background. Therefore it is better to represent Time complexities instead of reporting actual runtime cost as Time complexities normalizes external influence (like speed of the processor, availability of CPU).

7) Why do we analyze theoretical runtimes for algorithms instead of implementing them and reporting actual/experimental runtimes? Are there times when theoretical run- times provide more useful comparisons? Are their times when experimental runtimes provide more useful comparisons?

We cannot always compute actual/experimental runtimes cost for all the values of n. It would take a lot of time to do this. To avoid this, we generalize the Time and Space complexities using theoretical runtime. This simplifies assessing the performance costs of different algorithms.

Additionally theoretical run- times provide more useful comparisons by removing input biases and influence of external factors (like speed of the processor, availability of CPU).

Incase of a fixed input size and standard operating conditions (like same machine and load is uniform), experimental runtimes can provide useful comparisons than theoretical run- times.