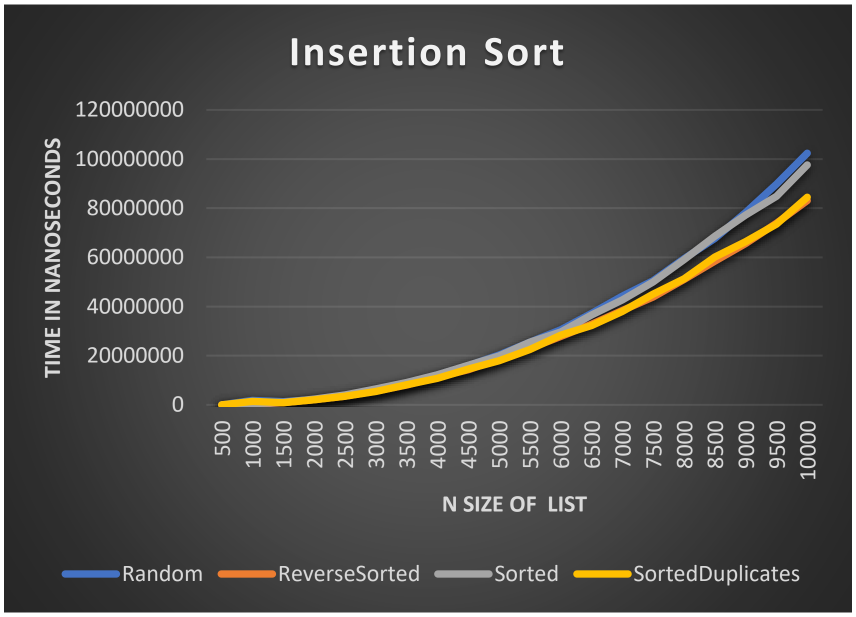
Harrison Smith — Assignment 05 — Prof. Joshi

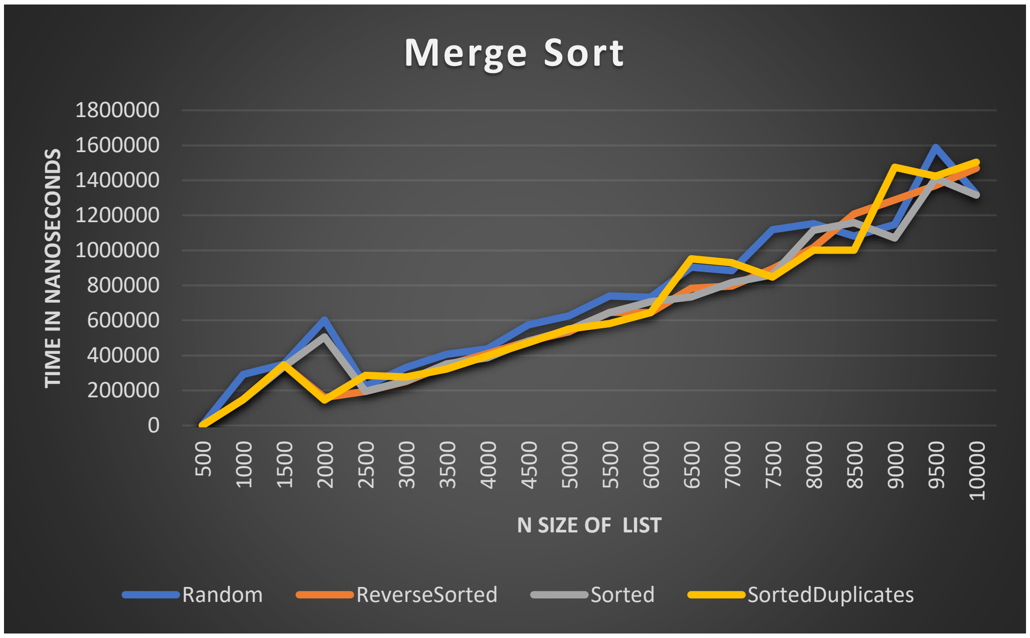
Samuel Hancock — Sorting Comparisons and Optimizations — October 3, 2019

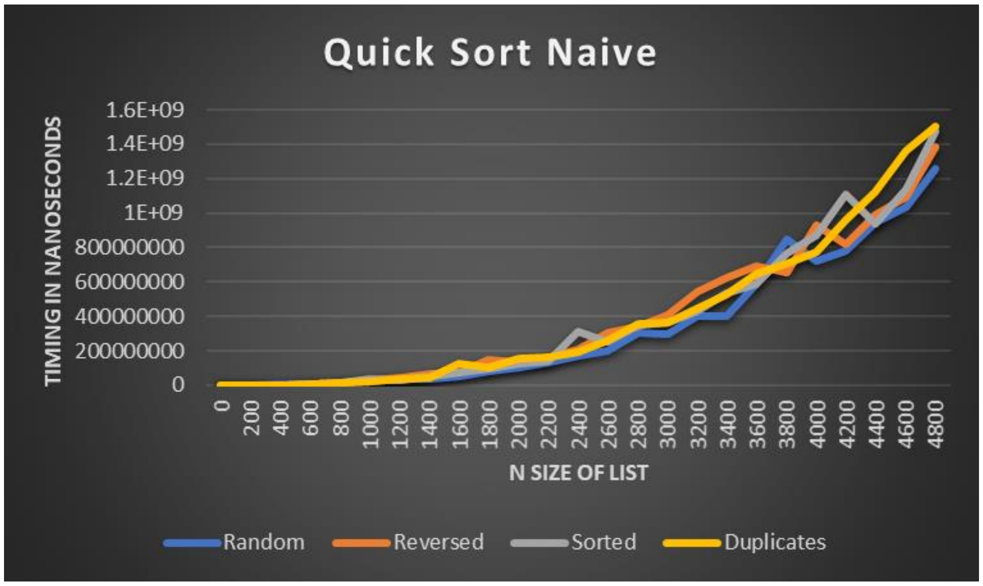
1a. Below are the graphs that will show the performance of various routines of the Sorting Algorithms



The complexity for Insertion sort is N^2 this is because Insertion Sort will require to evaluate element to each and every other element

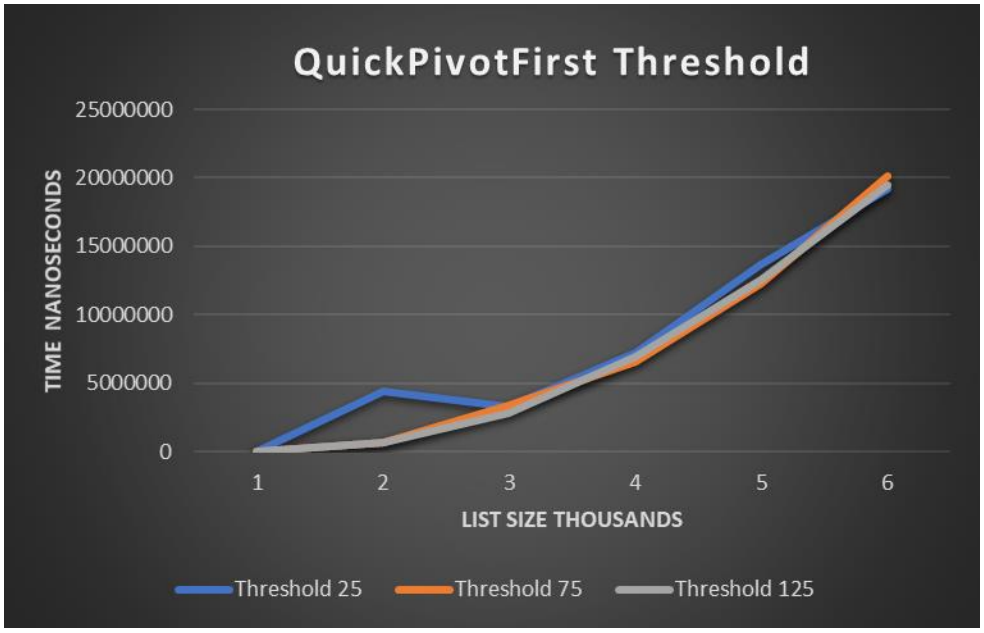
The complexity for Merge Sort has the complexity of NLogN because it is able to touch the element once with ability to forget about one side of the set since it is already partially sorted when it merges the split listed



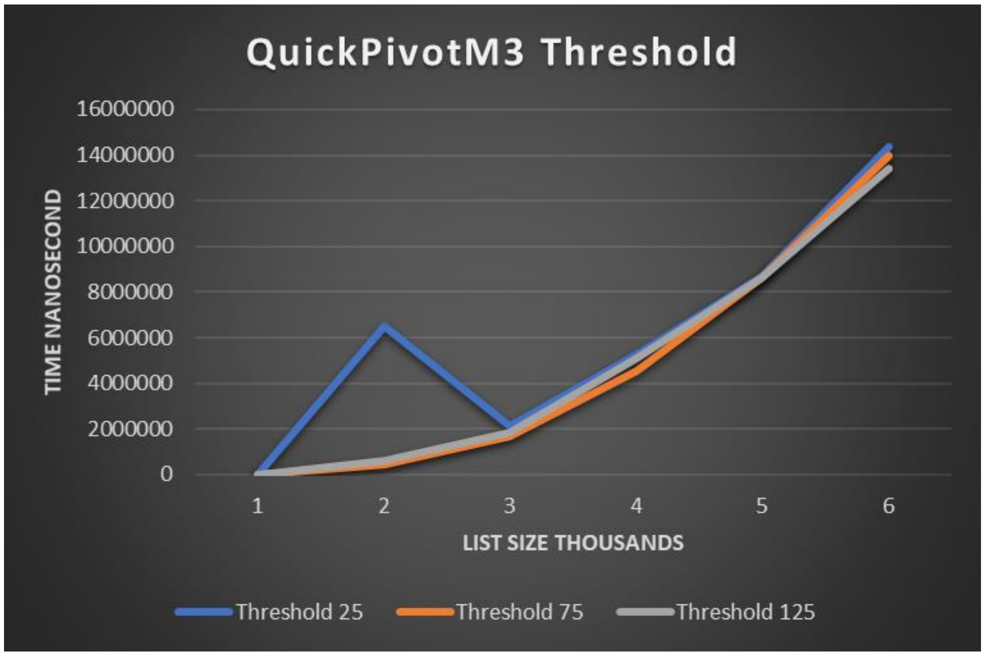


Quick Sort Naïve, is the worst possible choice each time it sorts and chooses a pivot, therefore, the runtime for this code will be N^2 since it will have to compare each element to each other element

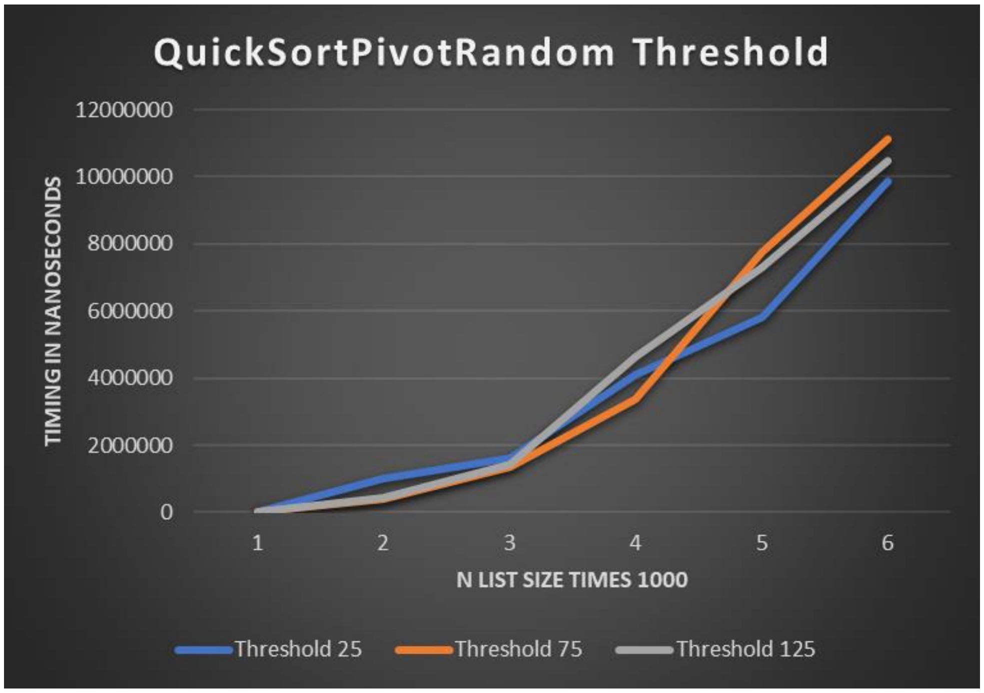
Below are some graphs after we implemented the threshold to help with run time, we will address each case directly.



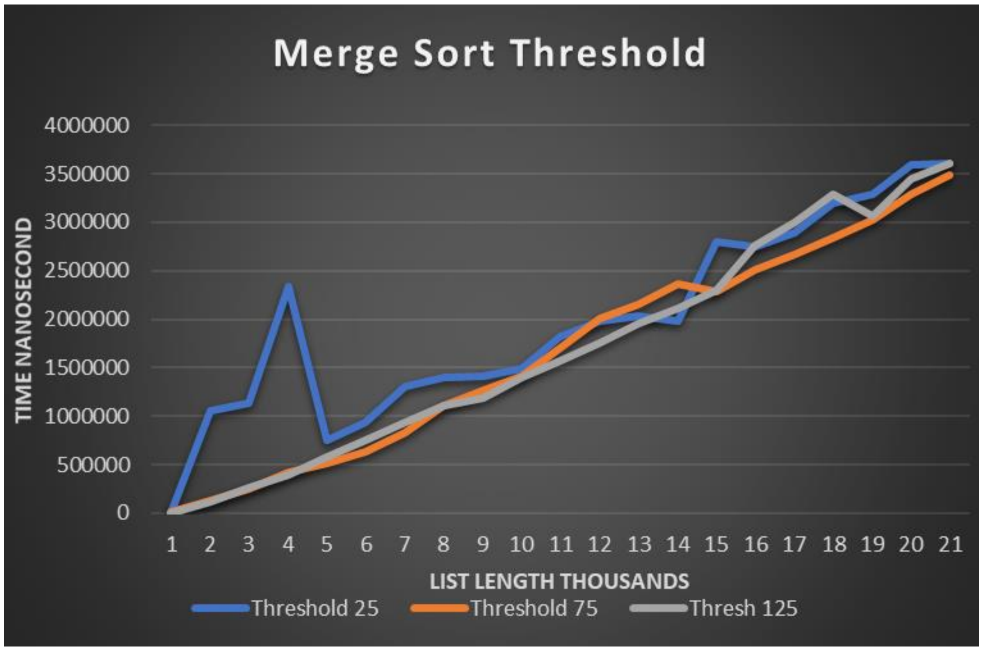
Comparing QuickPivotFirst with the threshold increasing, the larger it goes the faster it will sort the list because it will utilize the insertionSort, where insertionSort is efficient up to 125 elements



QuickSortM3 when applied the threshold will see a large impact on it’s timing of small elements, allowing it to perform more efficiently and sort out the data quickly



QuickSortPivotRandom is able to maintain a closely efficient runtime, however, it is able to improve still by utilizing the insertionSort's efficiency of 125 elements



However, we see the most impact for small sizes of lists, however, overall it still will perform best with the threshold remaining near the maximum of 125 threshold.

Harrison Smith

Samuel Hancock

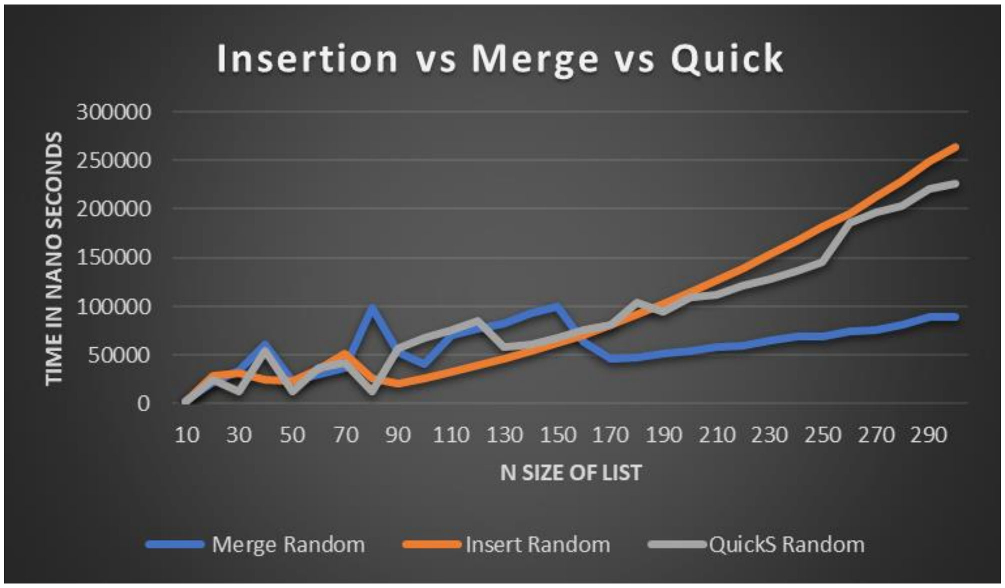
Assignment 05

Sorting Comparisons and Optimizations

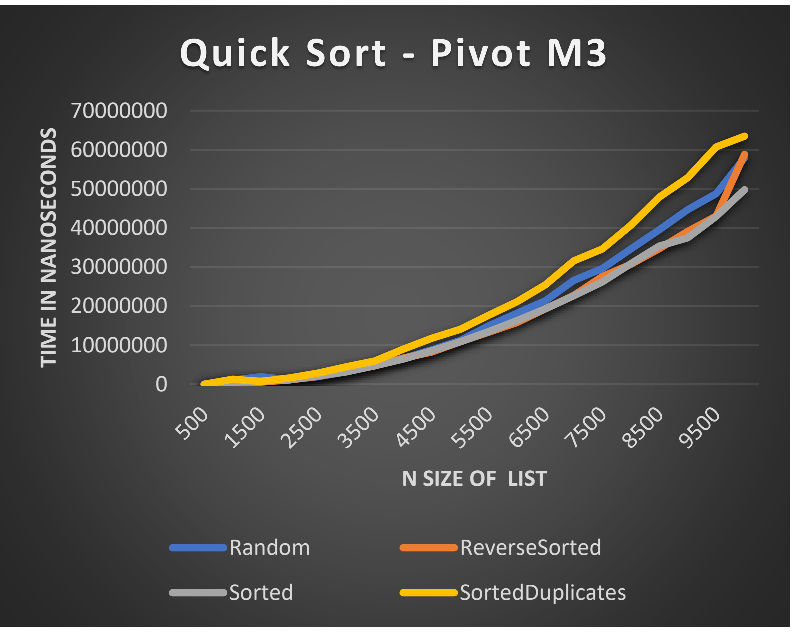
Prof. Joshi

October 3, 2019

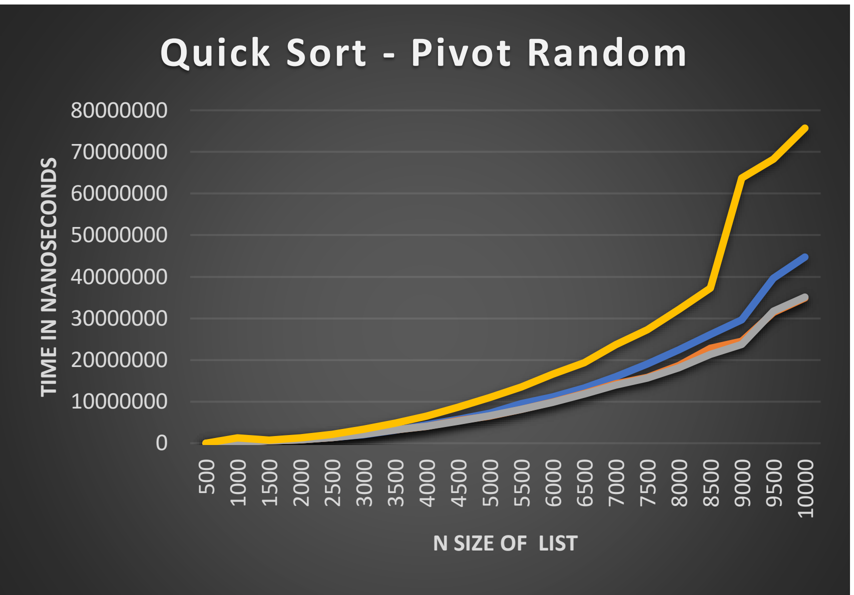
1b. Insertion Sort vs Merge Sort vs Quick Sort – the threshold we will want to use to optimize the sorting time is a threshold of 150 for insertion sort, once it hits 150 we will switch to merge sort since the timing for insertion is faster than the merge sort for a size of 150 or less



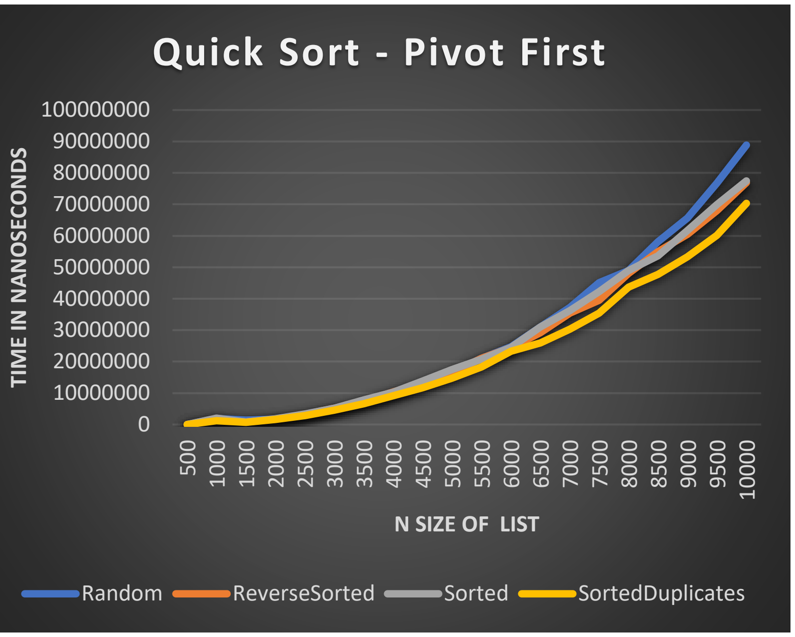
1c. Quick Sort Comparisons – Listed below are the graphs for the different implementations of the pivot location for Quick Sort

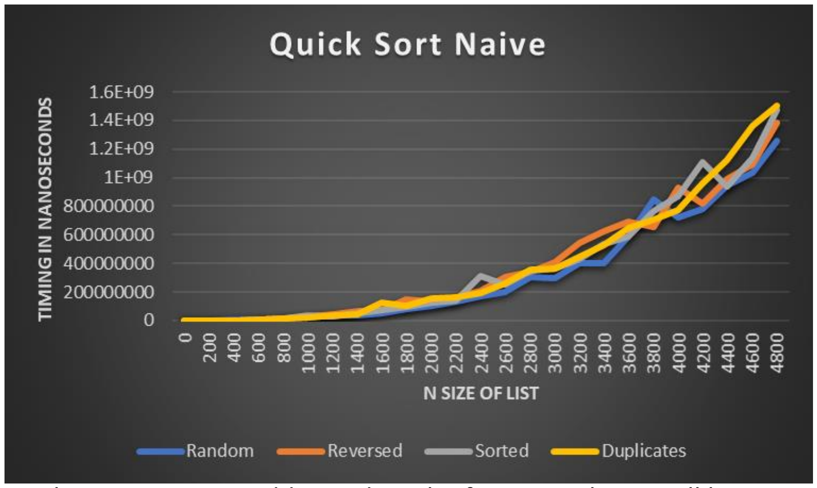


The complexity of M3 for most instances will be NLogN because we will be able to split what is being sorted by pivots and choosing the correct location to bring comparison to



Quick Sort Pivot Random will retain closer to NLogN because it will have a smaller possibility of choosing a bad starting pivot, avoiding the worst run time of N^2

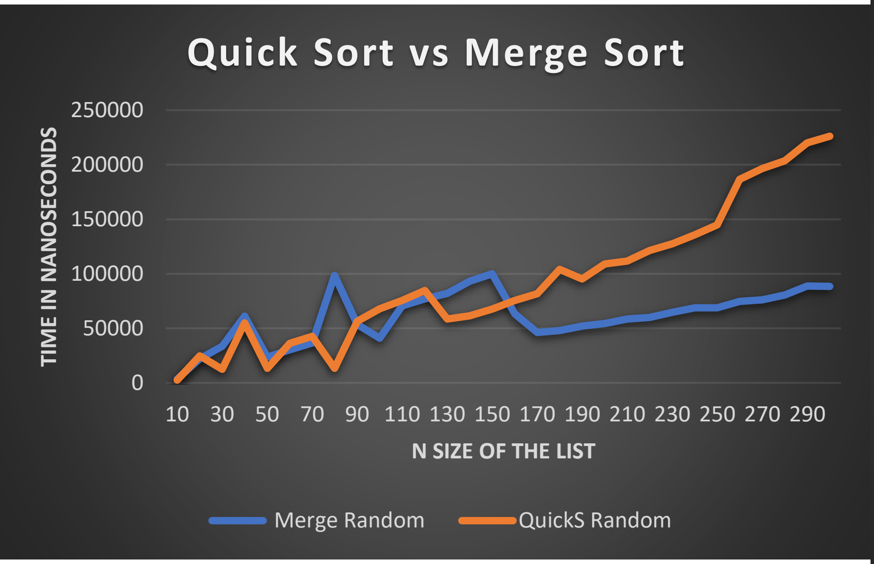


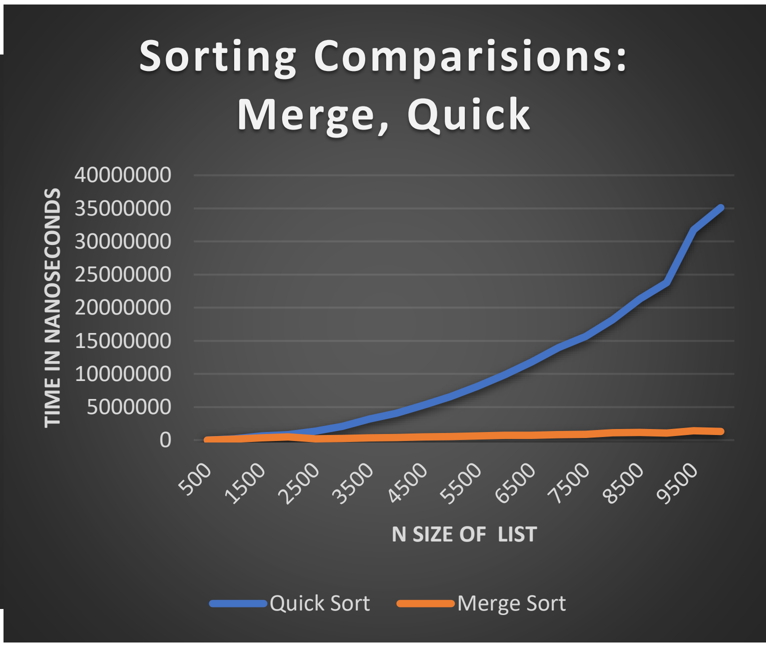


QuickSort Naïve was addressed on the first page, but it will have a similar runtime to Quick Sort First, because it will not be able to determine a good element location

Quick Sort Pivot First will always pick the worst possible runtime, this is because it will start at the beginning and have to compare each item to each other item

1d. There are two graphs to compare the difference in time for Quick Sort and Merge Sort, one close up and one with a larger size of N





Quick Sort performs more efficiently than merge sort for the first 150 items, the it will run into cases where it will perform at it's worst efficiency of N^2 instead of NlogN

2. For a small list the pivot point would exceed the values inside the list, so there wouldn't be anything to swap around because the pivot point might be outside. Additionally, it would leave a permanent sorting position of the pivot, the loop would iterate once, find the correct location of that specific value of the pivot, if it was contained in the list, and keep looping over the list, not changing anything.

3a. We have learned that it is important to understand and avoid making multiple objects to pass the values through to keep our performance faster, thus having a better run time overtime and not duplicating data or running into a stack overflow due to a large usage of memory. Therefore, if we are going to be modifying lists and keep the helper lists or helper objects to a minimum like we were able to achieve in merge under merge sort, where we were able to improve upon our first implementation from passing in two lists and merging to a single sorted list then overriding to the first list; now we are able to utilize indices to assist in keeping our lists to one helper list that keeps the sorted data until we are ready to set it in the original list.

3b. Finally, we have learned that it is important to understand the runtime of our algorithms, since there are certain sizes where other algorithms will perform more efficiently then the algorithm that has the better overall runtime; insertion sort performs faster than merge and quick sort on lists sizes of 150 or less, so we would set a threshold to use insertion sort when sorting on lists to utilize this performance enhancement. Moreover, we should carefully analysis all of our algorithms and see if they are able to out preform each other at certain points.