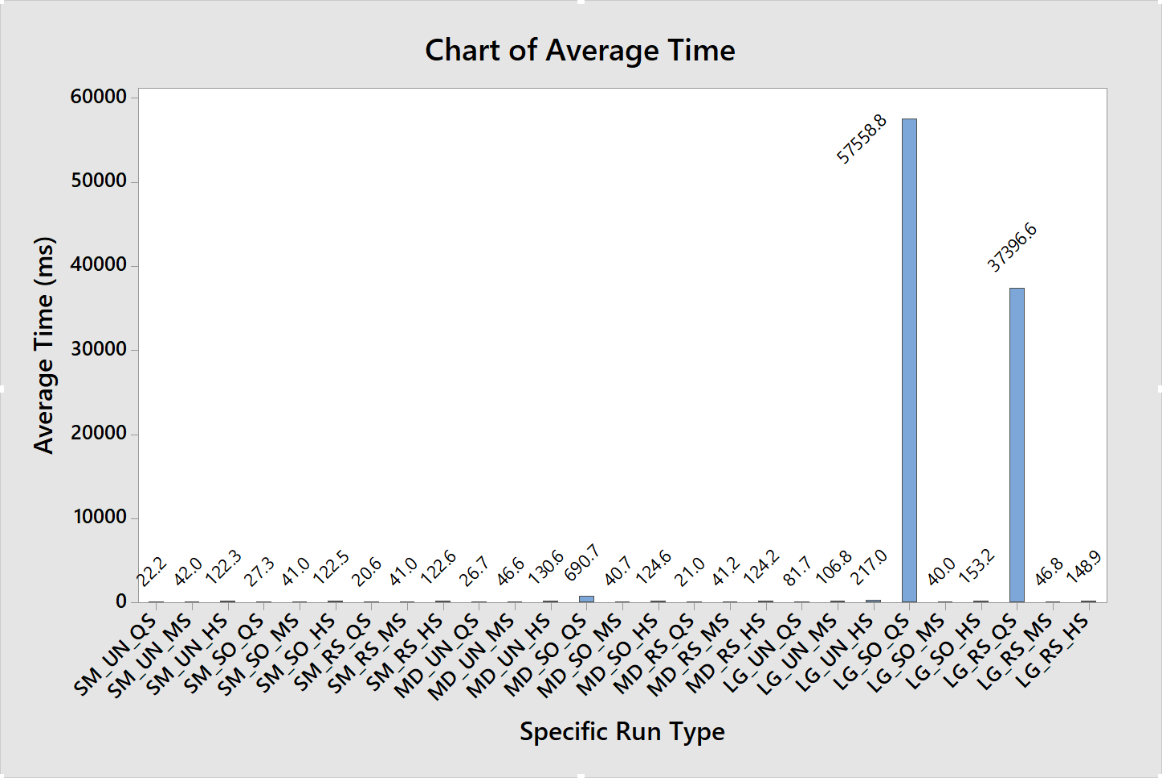
Adam Pine

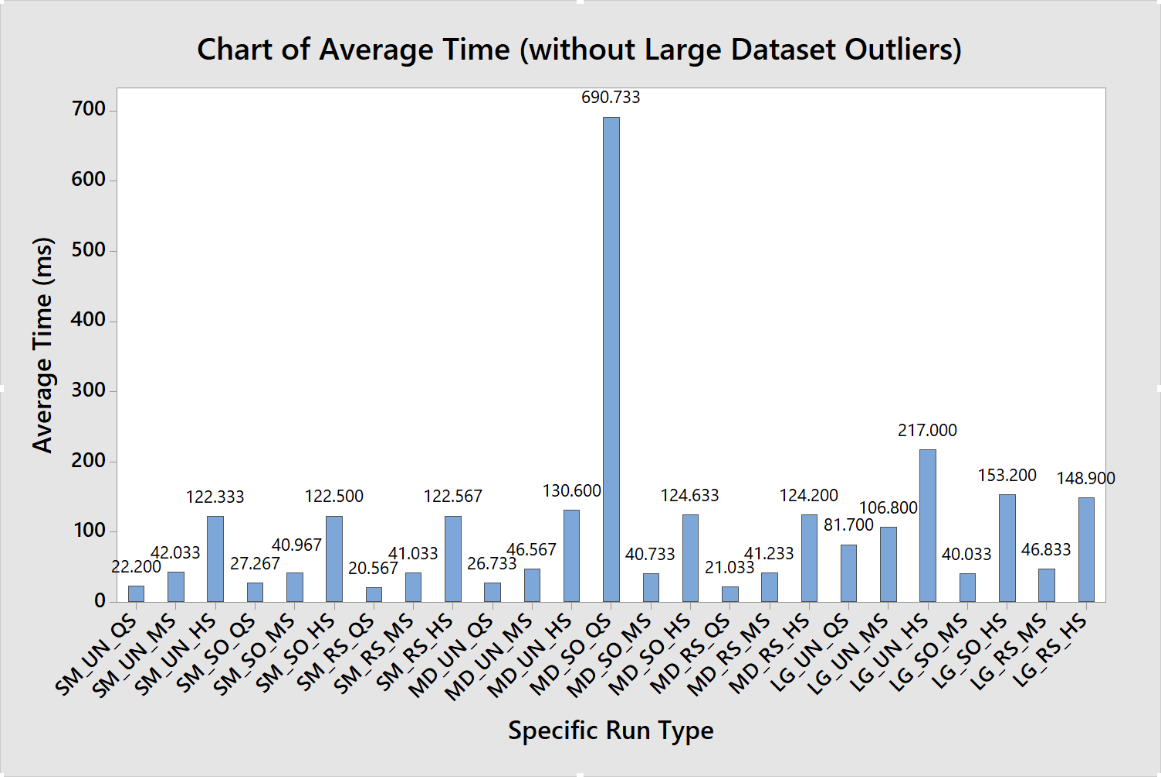
Sorting Algorithms Implementation and Analysis

With my implementations, I found that QuickSort was the fastest of the different algorithms, for most cases. That is, that it performed well on unsorted data, but QuickSort took a very long time, when I gave it a pre-sorted dataset, Seen below:



If we remove the 2 large outliers, we see that QuickSort exhibits the same issues with the same sort type, in the Medium sized datasets (100,000 items).

This can be seen in “Chart of Average Time (without Large Dataset Outliers)”

Using this data, if we ignore the sizes of the files, we can see that merge sort is the most consistent in performance, while QS is typically the fastest, as long as the input is not already sorted.

We get this table, if we ignore the sizes of the files: (data is the average time in ms, but removing the dataset size filtering criteria (getting rid of sm/md/lg))

|  |  |  |  |
| --- | --- | --- | --- |
| Presort Type / Sort Algorithm | QuickSort | MergeSort | HeapSort |
| Unsorted | 43.5 ms | 65.1333 ms | 156.644 ms |
| Sorted | 19425.6 ms | 40.5778 ms | 133.444 ms |
| Reverse sorted | 12479.4 ms | 43.0333 ms | 131.889 ms |

Obviously, we have to look at this table with a grain of salt, as we are disregarding specific criteria that was used when the data was recorded. Overall, QuickSort is the fastest algorithm to use, unless your dataset is likely to already be in order. If so, use MergeSort, or even HeapSort. This outcome is also confirmed be the standard deviations, where most of the stand deviations are between 0 and 6. But, when we look at the ones for Sorted, Quick Sort runtimes, we see that once we get to the medium sized dataset, the standard deviation is starting to increase by a lot, because the standard deviation for the MD\_SO\_QS dataset is 31 (see Minitab project that is in the attached zip). This is even further noticeable when we look at the Large, Sorted, QuickSort runtimes, the standard deviation is 13098.3 ms, which is a huge discrepancy between each run. This same issue wasn’t present previously, but once we get to the large dataset, the reverse sorted dataset also had a large outlier for standard deviation, at 148.1 ms which is better, but still very high. These runtime extremes are due to that fact that I did not add in any kind of efficiency-optimization in some of the different actions.

Another test that might be useful is to see how much these speeds change, if the datasets are only slightly-ordered, and how that might affect the runtime of QS on the large datasets.

Comparing to the worth case performance for quicksort, this kind of makes sense, as the quicksort is doing a 1,000,000 ^ 2 (1 trillion) operations on the dataset. In the average/best case, where the array is not already sorted, quicksort is easily one of the fastest algorithms.

Comparing MergeSort to its worst case, n log n, it looks as if my implementation is working fairly well. Looking at my HeapSort performance, it seems that it may not be the most efficient implementation, as it should have about the same runtime as my mergesort, as their worst-case performances are both n log n. This may be due to my use of a BottomUpHeapsort algorithm, without using the correct type of pivot selection.

Conclusion: If you need to sort a dataset, use quicksort, as long as it is not already sorted, otherwise choose MergeSort, or HeapSort (assuming you also make it more efficient than what I was able to).

I will attach the code to d2l, but you can find the code for this project on github, here: <https://github.com/Adondriel/AlgorithmsProject>

Info for minitab project:

Most data is in Worksheet 2. Worksheet 4 has the none-size groups data in it.