Project 1: Sorting and Matrix Multiplication

CS331.01

Michael Choi

2/19/18

Introduction:

For this project, the purpose was to implement difference sorting and matrix multiplication algorithms to see the difference is running time. Although all the sorting algorithms and both matrix multiplication algorithms achieved the same goal, the difference in speed was very significant. For testing, various sizes were chosen and random numbers ranging from 1 to n were generated into the array.

Analysis:

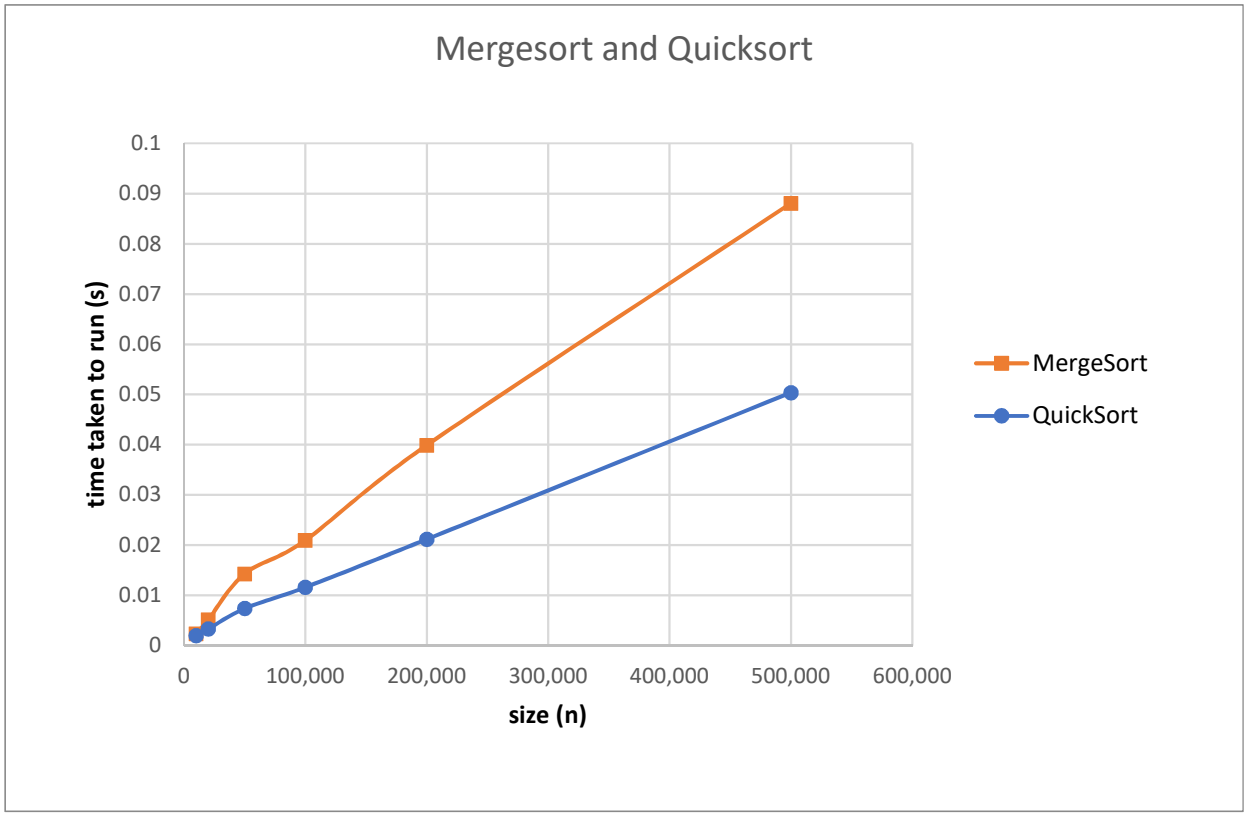
Sorting:

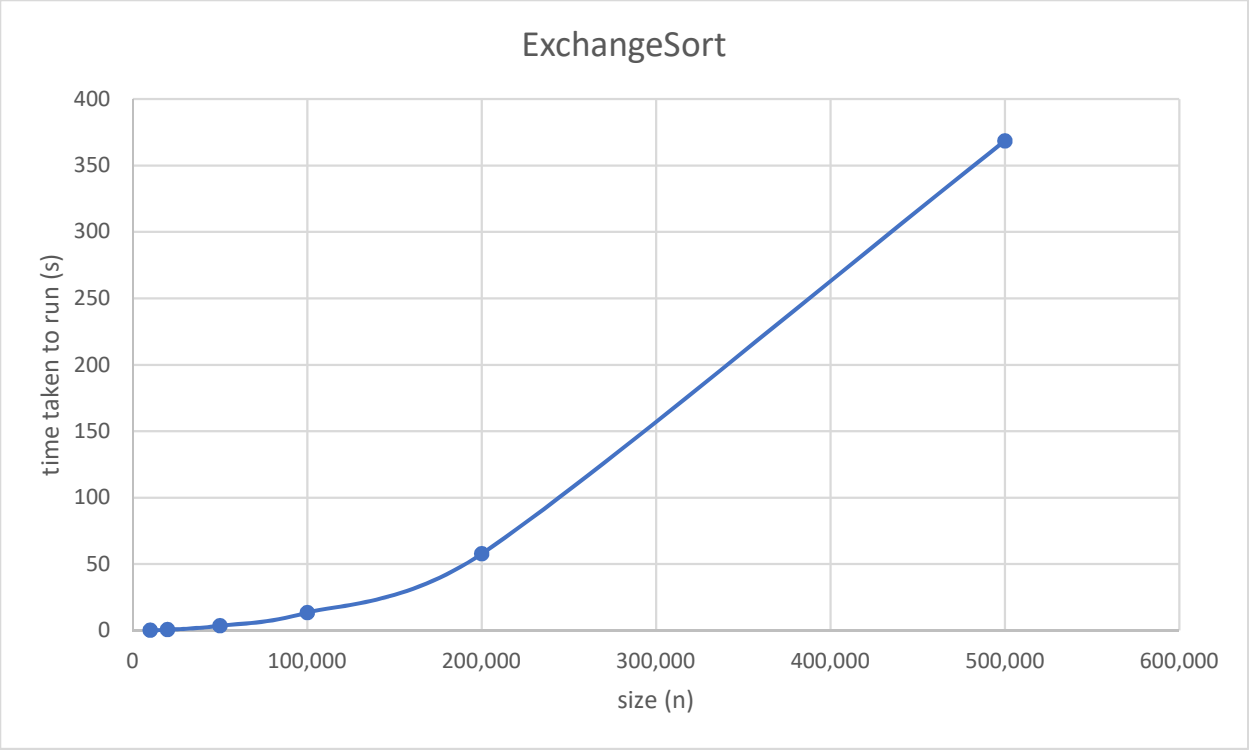
As expected exchange sort took significantly longer than both merge and quick sort. Surprisingly, quick sort took less time on all array sizes compared to merge sort. Here is a table comparing the times in seconds based on various arrays of size n:

	MergeSort	QuickSort	ExchangeSort
n = 10,000	0.002350916	0.001983721	0.147482263
20,000	0.005153764	0.003291096	0.598267759
50,000	0.014297008	0.007375867	3.608379119
100,000	0.020940969	0.011609626	13.480319071
200,000	0.039908459	0.021166934	57.739089922
500,000	0.088092924	0.050375839	368.572255281
1,000,000	0.177389863	0.098842762	x
10,000,000	1.848903832	1.069428998	x

100,000,000	20.039219386	11.643144057	x
200,000,000	41.205702353	24.60783438	x
300,000,000	x	36.690938048	x
350,000,000	x	43.908899368	x

Although theoretically merge sort should be the fastest, in my implementation, quick sort outperformed merge sort and was even able to get to higher entry length at 350,000,000 while merge sort had memory errors starting after 200,000,000. Here are some graphs to display the difference visually:





Quick and merge sort seem to have a similar trendline of nlogn while exchange sort seems more to look like n^2. Because the difference between time ran of merge and quick compared to exchange was so great that fitting all lines on one graph wasn't feasible without making quick and mergesort look like linear lines. One reasons as to why quicksort might be faster when implemented is that not all processes take the same time and merge sort has a lot more allocating new arrays and merging compared to quicksort. Exchange sort as expected is very inefficient and feels slow starting at around 50,000 entries.

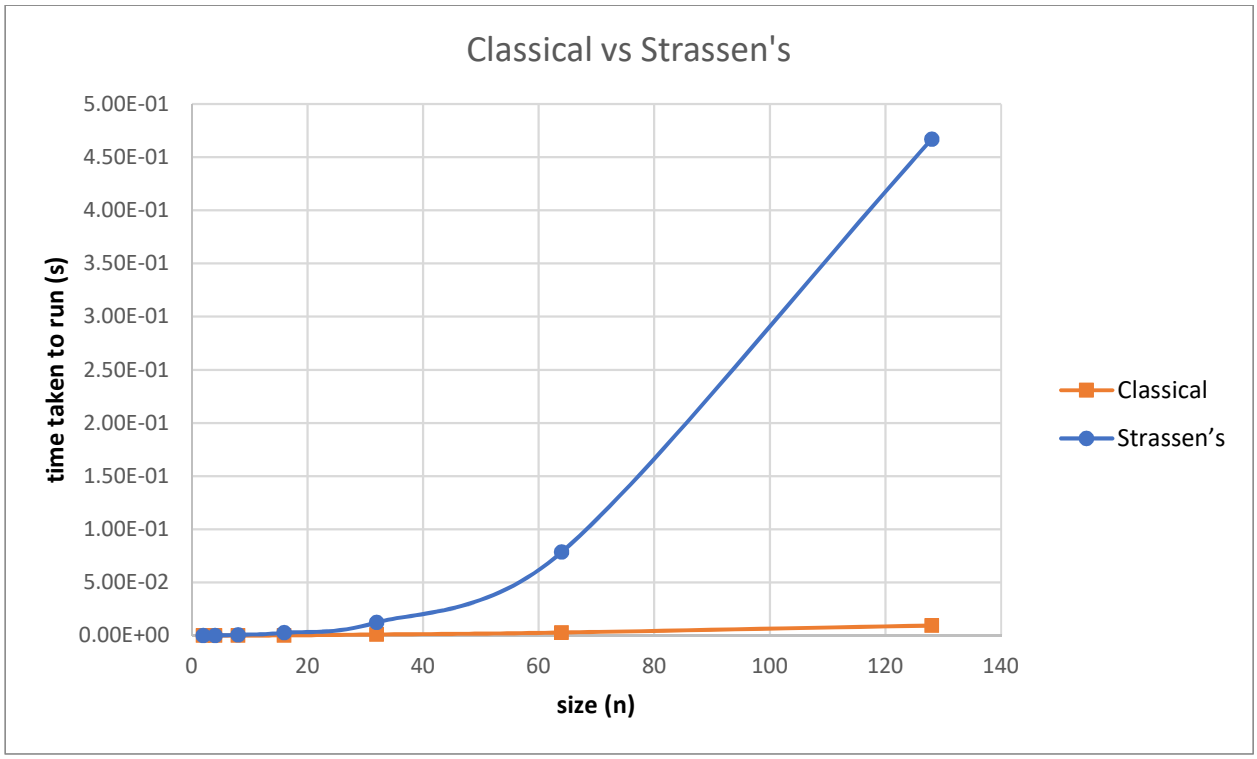
Matrix Multiplication:

There was another surprising result for the matrix multiplication part. Classical multiplication beat out Strassen's algorithm by a large margin for every size we tried. It is known that Strassen's is not effective when using small n values, but even at 2048 classical was hugely better. This might mean that Strassen's needs a much larger value of n to outperform

classical, but I couldn't test higher than 2048 for Strassen's and 4096 for classical due to the overwhelming amount of time and memory limitation.

	Classical	Strassen's
n = 2	4.656E-6	2.2659E-5
4	6.828E-6	1.03981E-4
8	2.1107E-5	6.9466E-4
16	1.30986E-4	0.002622199
32	9.70599E-4	0.012338429
64	0.002812781	0.07848316
128	0.009372625	0.467049463
256	0.025960022	2.909202623
512	0.251606982	20.232578344
1024	3.049913354	143.475329816
2048	114.778179874	986.770569661
4096	1032.70566464	x

Here is also the graph comparing the two methods visually:



The implementation seems to take a huge role in the time complexity for matrix multiplication as well. Strassen's might take a lot longer due to dynamically allocating arrays quite frequently while the classical method only needs matrix a, matrix b, and product matrix c to run the whole program.