Ryan Ellis
COSC 220-003
Homework #8: Timing Analysis of Faster Sorts
11/21/2024

Recorded Output Observations

Integer Sort Timing

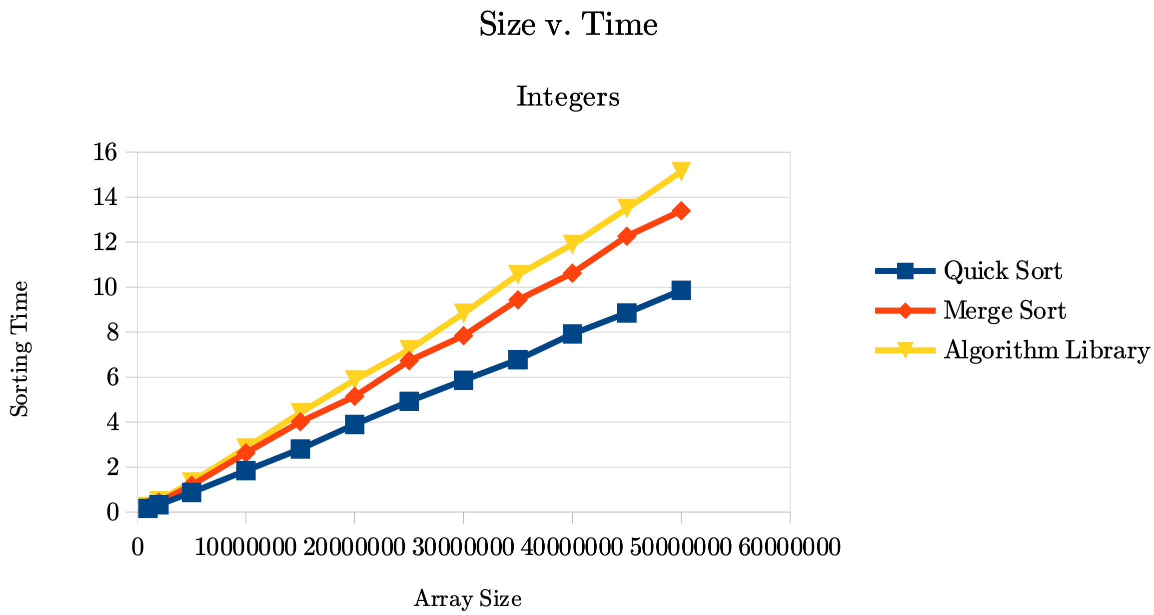

Figure 1: Timing of Sorting Integers

Table 1: Recorded Output Data

Integers	Quick Sort	Merge Sort	Algorithm Library
1000000	0.161351	0.227108	0.252293
2000000	0.328769	0.445868	0.506061
5000000	0.870329	1.19392	1.33873
10000000	1.84134	2.63208	2.85286
15000000	2.80114	4.01576	4.41503
20000000	3.8928	5.14584	5.87331
25000000	4.92254	6.72955	7.21993
30000000	5.85774	7.84057	8.83853
35000000	6.77387	9.43357	10.5545
40000000	7.91911	10.6222	11.8982
45000000	8.84967	12.2591	13.4923
50000000	9.85659	13.3924	15.1351

Analysis of Integer Sort Timing

Upon reviewing the data recorded from the main program output tests for the integer selections, the three algorithms are relatively within the same margins with the exception of **Quick Sort**, which provides a more computational and time effective result compared to the **Merge Sort** and the **Algorithm Library** sorting method. Looking at the results graphically, all three methods have the same linear shape, but vary in amplitude given their respective efficiency costs. Furthermore, the structure of each algorithm is indicative of their prospective results, that is, **Quick Sort** uses local resources and recursion to sort which correlates with the low computation cost and time efficiency, whereas **Merge Sort** uses pointers (additional dynamic memory) and the **Algorithm Library** has scalability for multiple cases but falls short compared to **Quick Sort**.

Double Sort Timing

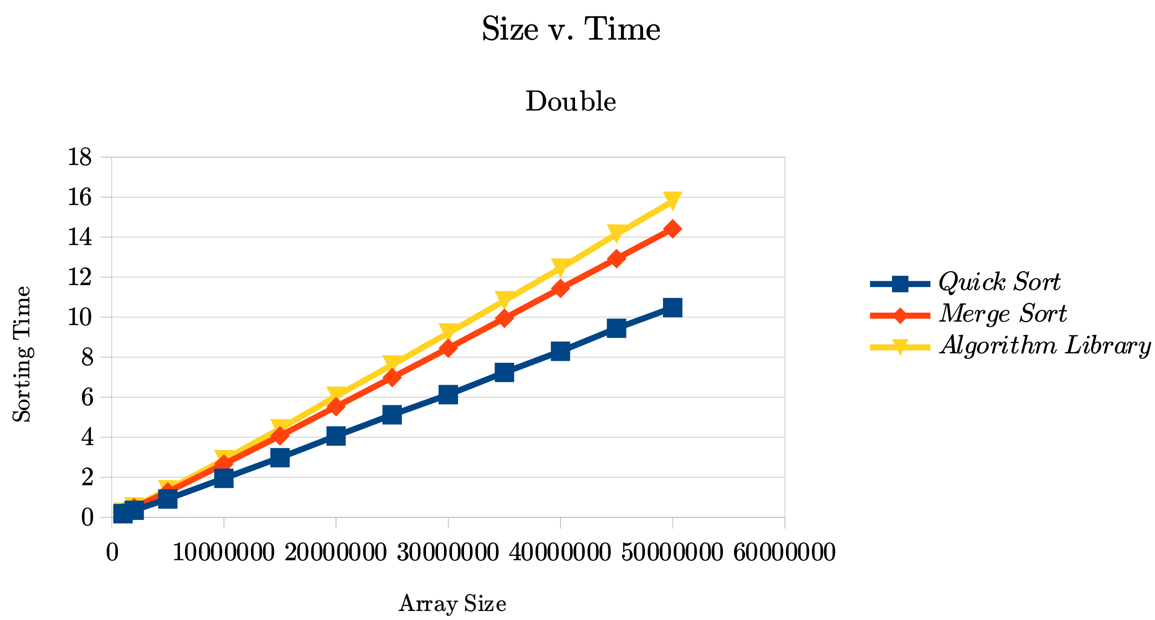

Figure 2: Timing of Sorting Doubles

Table 2: Recorded Output Data

Doubles	Quick Sort	Merge Sort	Algorithm Library
1000000	0.17856	0.237136	0.25077
2000000	0.349967	0.481671	0.521784
5000000	0.913608	1.26447	1.37521
10000000	1.94754	2.66384	2.88156
15000000	2.9759	4.06396	4.42229
20000000	4.05975	5.5294	6.05005
25000000	5.12146	6.98049	7.61085
30000000	6.1296	8.44738	9.20471
35000000	7.23508	9.94328	10.8257
40000000	8.29392	11.438	12.4505
45000000	9.44454	12.9221	14.1514
50000000	10.4708	14.4155	15.7925

Analysis of Double Sort Timing

Looking at the results for double array sorting, it can be noted that the results are very similar in comparison to the integer array sorting. Moreover, the three algorithms are relatively within the same margins with the exception of **Quick Sort**, which provides a more computational and time effective result compared to the **Merge Sort** and the **Algorithm Library** sorting method. Looking at the results graphically, all three methods have the same linear shape, but vary in amplitude given their respective efficiency costs. Furthermore, the structure of each algorithm is indicative of their prospective results, that is, **Quick Sort** uses local resources and recursion to sort which correlates with the low computation cost and time efficiency, whereas **Merge Sort** uses pointers (additional dynamic memory) and the **Algorithm Library** has scalability for multiple cases but falls short compared to **Quick Sort**.

Merge Sort Comparison Plot

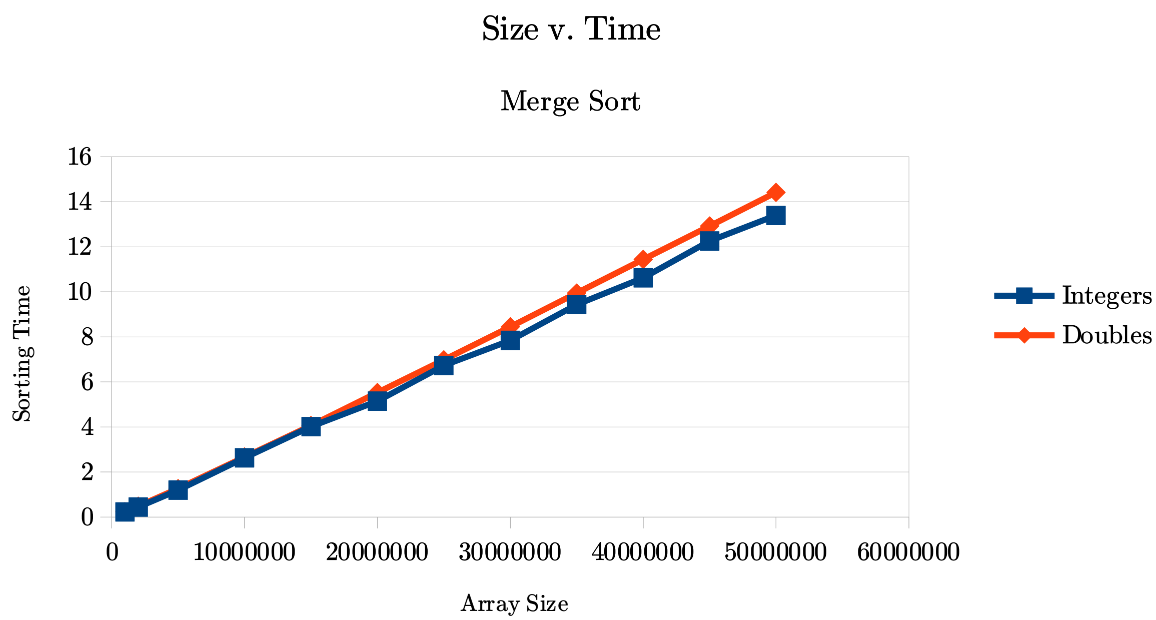

Figure 3: Merge Sort comparison of Integers and Doubles

Table 3: Recorded Output Data

Merge	Integers	Doubles
1000000	0.227108	0.237136
2000000	0.445868	0.481671
5000000	1.19392	1.26447
10000000	2.63208	2.66384
15000000	4.01576	4.06396
20000000	5.14584	5.5294
25000000	6.72955	6.98049
30000000	7.84057	8.44738
35000000	9.43357	9.94328
40000000	10.6222	11.438
45000000	12.2591	12.9221
50000000	13.3924	14.4155

Analysis of Data Types with Merge Sort

In comparison to sorting of integer and double values, it can be noticed that there is not a distinct level of marginal difference between the initial array sizes between the data types. However, as the size of the array increases, so does the latency in computation efficiency between the two data types. This can be attributed to the timing of comparison between integer values and double values, where double values can store values that require more memory to analyze in comparison to integer values. Looking at the results graphically, both data types have the same linear shape, but vary in amplitude given their respective efficiency costs.

Quick Sort Comparison Plot

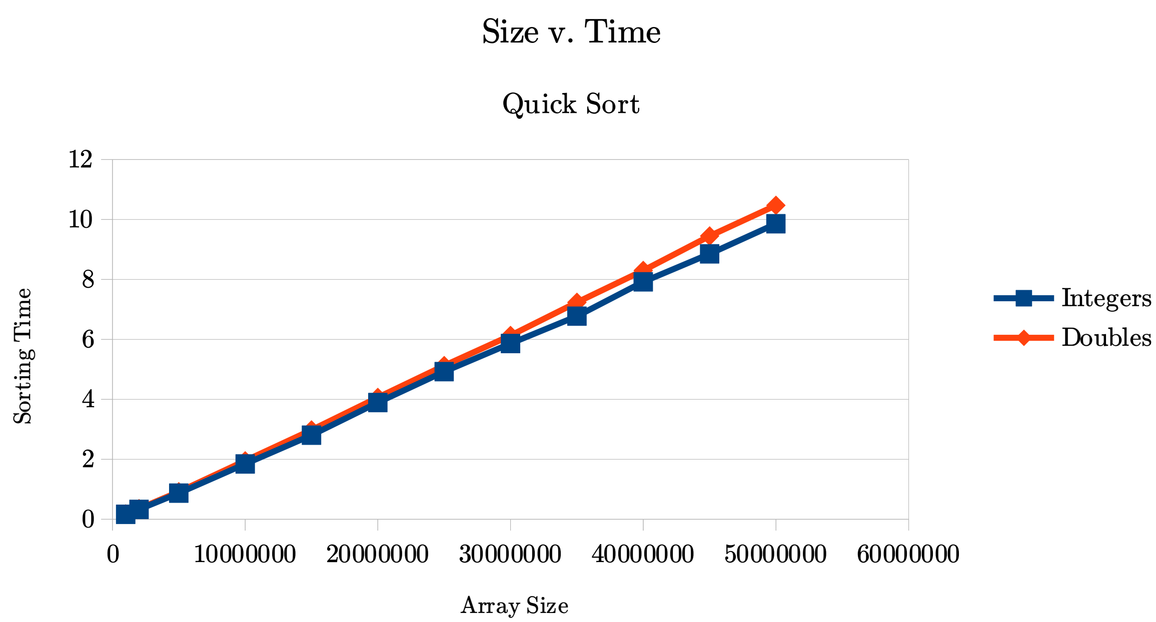

Figure 4: Quick Sort comparison of Integers and Doubles

Table 4: Quick Sort comparison of Integers and Doubles

Quick Sort	Integers	Doubles
1000000	0.161351	0.17856
2000000	0.328769	0.349967
5000000	0.870329	0.913608
10000000	1.84134	1.94754
15000000	2.80114	2.9759
20000000	3.8928	4.05975
25000000	4.92254	5.12146
30000000	5.85774	6.1296
35000000	6.77387	7.23508
40000000	7.91911	8.29392
45000000	8.84967	9.44454
50000000	9.85659	10.4708

Analysis of Data Types with Quick Sort

Similar to the **Merge Sort** table and graph, the **Quick Sort** method shows a similar shape in the graph, but a faster sorting time. Also, it can be noticed that there is not a distinct level of marginal difference between the initial array sizes between the two data types. Again, as the size of the array increases, so does the latency in computation efficiency between the two data types. This can be attributed to the timing of comparison between integer values and double values, where double values can store values that require more memory to analyze in comparison to integer values. Looking at the results graphically, both data types have the same linear shape, but vary in amplitude given their respective efficiency costs.

Algorithm Library Sort Comparison Plot

Size v. Time

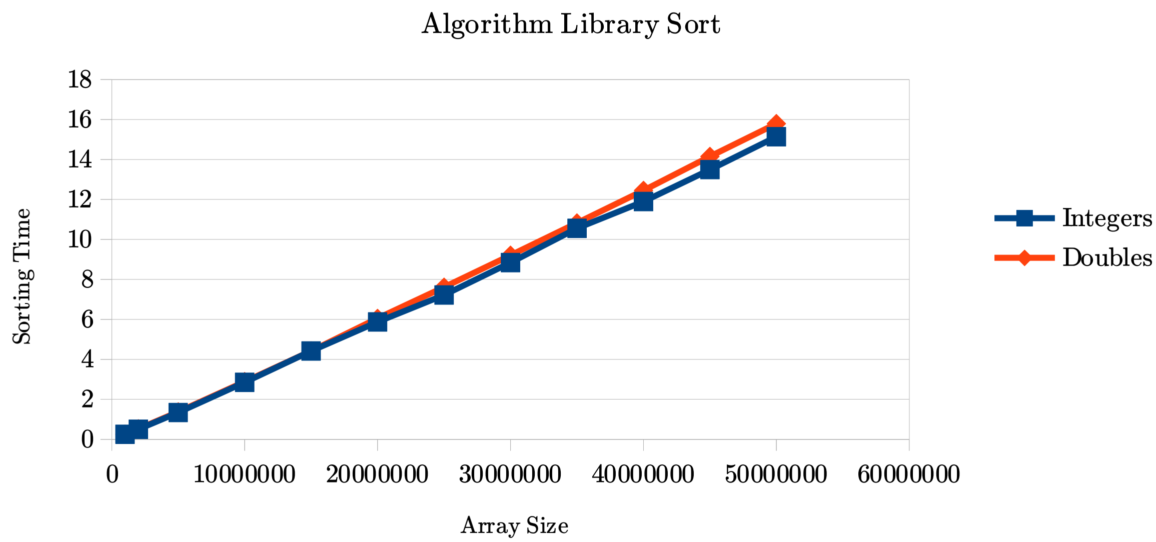

Table 5: Algorithm Sort comparison of Integers and Doubles

Algorithm Sort	Integers	Doubles
1000000	0.252293	0.25077
2000000	0.506061	0.521784
5000000	1.33873	1.37521
10000000	2.85286	2.88156
15000000	4.41503	4.42229
20000000	5.87331	6.05005
25000000	7.21993	7.61085
30000000	8.83853	9.20471
35000000	10.5545	10.8257
40000000	11.8982	12.4505
45000000	13.4923	14.1514
50000000	15.1351	15.7925

Analysis of Data Types with Algorithm Library Sort

Similar to the **Merge Sort** table and graph, the **Algorithm Library Sort** method shows a similar shape in the graph, and a comparable sorting time. Also, it can be noticed that there is not a distinct level of marginal difference between the initial array sizes between the two data types. Again, as the size of the array increases, so does the latency in computation efficiency between the two data types. This can be attributed to the timing of comparison between integer values and double values, where double values can store values that require more memory to analyze in comparison to integer values. Looking at the results graphically, both data types have the same linear shape, but vary in amplitude given their respective efficiency costs.

Conclusion

Analyzing all of the gathered data and tables, it is clear that the **Quick Sort** method is more time efficient and computationally resourceful in comparison to the other methods of sorting. Given the margin is relatively small within this data set, the pattern is clear that as the data set grows, so does the margin of efficiency for the **Quick Sort** method. With regards to data types (integers v. doubles), the **Quick Sort** method also shows a significant difference in speed for both data types, with a slight damper with regards to sorting doubles. This can be attributed to the amount of memory it takes for a floating point value to be stored in comparison to an integer, which holds true to the findings of this data.