ANALYSIS OF SORTING ALGORITHMS

March 19, 2018

Scott McCoy
108786190

INTRODUCTION

The goal of this project was to learn about several different sorting algorithms and the relationship between their time complexities and their real world run times. This was achieved by implementing each algorithm in C++ and observing, charting, and comparing the changes in run times for each algorithm when using different lengths and configurations for the array. A statistical analysis of the run times of each configuration and algorithm was also performed.

METHODS

The algorithms that were implemented were the insertion sort, selection sort, bubble sort, merge sort, and quick sort. Each sorting algorithm was ran with arrays of lengths 10, 1000, 10,000, 100,000, and 1,000,000. The four different configurations for the arrays were already sorted, randomly shuffled, sorted in reverse, and shuffled at 10 percent. Each combination of array length and configuration was sorted 100 times using each algorithm to produce the data for the statistical analysis. A menu system to allow a user to choose the type of sort, the length of the array, and the type of the array was also implemented.

RESULTS

The results of the tests are represented by the following tables. The cutoff that was chosen for the runtime was 90 minutes meaning that if the test ran for more than 90 minutes then it would be stopped for the sake of time. You may notice that some tests were ran for longer but this was only because I was unaware that stopping the test early was acceptable.

Legend

Symbol	Meaning
X	Execution of tests took longer than 90 minutes
Case 1	Already Sorted
Case 2	Randomly Ordered
Case 3	Sorted in Reverse
Case 4	Shuffled at 10%

Table 1 - Bubble Sort Results

Case	Average Time (seconds)	Standard Deviation
Size 10 Case 1	2.73 * 10^-7	3.305 * 10^-6
Size 10 Case 2	4.02 * 10^-6	5.85 * 10^-7
Size 10 Case 3	3.32 * 10^-6	6.01 * 10^-6
Size 10 Case 4	2.87 * 10^-6	2.29 * 10^-6
Size 1000 Case 1	2.74 * 10^-3	0.000347
Size 1000 Case 2	3.98 * 10^-3	0.000394
Size 1000 Case 3	4.84 * 10^-3	0.000355
Size 1000 Case 4	2.96 * 10^-3	0.000389
Size 10000 Case 1	2.71 * 10^-1	0.00568
Size 10000 Case 2	4.59 * 10^-1	0.00131
Size 10000 Case 3	4.76 * 10^-1	0.00056
Size 10000 Case 4	2.84 * 10^-1	0.00046
Size 100000 Case 1	2.69 * 10^1	0.5802
Size 100000 Case 2	5.03 * 10^1	0.2011
Size 100000 Case 3	4.73 * 10^1	0.0297
Size 100000 Case 4	2.96 * 10^1	0.2205
Size 1000000 Case 1	X	X
Size 1000000 Case 2	X	X
Size 1000000 Case 3	X	X
Size 1000000 Case 4	X	X

Table 2 - Selection Sort Results

Case	Average Time (seconds)	Standard Deviation
Size 10 Case 1	2.73 * 10^-6	3.305 * 10^-6
Size 10 Case 2	4.02 * 10^-6	5.856 * 10^-7
Size 10 Case 3	3.32 * 10^-6	6.010 * 10^-7
Size 10 Case 4	2.87 * 10^-6	2.299 * 10^-6
Size 1000 Case 1	2.74 * 10^-3	0.000347
Size 1000 Case 2	3.98 * 10^-3	0.000394
Size 1000 Case 3	4.84 * 10^-3	0.000355
Size 1000 Case 4	2.96 * 10^-3	0.000389
Size 10000 Case 1	2.71 * 10^-1	0.005688
Size 10000 Case 2	4.59 * 10^-1	0.001312
Size 10000 Case 3	4.76 * 10^-1	0.000560
Size 10000 Case 4	2.84 * 10^-1	0.000469
Size 100000 Case 1	2.69 * 10^1	0.580209
Size 100000 Case 2	5.03 * 10^1	0.201183
Size 100000 Case 3	4.73 * 10^1	0.029740
Size 100000 Case 4	2.96 * 10^1	0.220500
Size 1000000 Case 1	X	X
Size 1000000 Case 2	X	X
Size 1000000 Case 3	X	X
Size 1000000 Case 4	X	X

Table 3 - Insertion Sort Results

Case	Average Time (seconds)	Standard Deviation
Size 10 Case 1	6.03 * 10^-7	1.984 * 10^-6
Size 10 Case 2	9.40 * 10^-7	2.616 * 10^-6
Size 10 Case 3	8.90 * 10^-7	1.984 * 10^-6
Size 10 Case 4	9.20 * 10^-7	2.111 * 10^-6
Size 1000 Case 1	4.34 * 10^-6	4.76 * 10^-7
Size 1000 Case 2	6.39 * 10^-4	1.82 * 10^-5
Size 1000 Case 3	1.30 * 10^-3	2.29 * 10^-5
Size 1000 Case 4	8.99 * 10^-5	6.61 * 10^-6
Size 10000 Case 1	3.95 * 10^-5	2.81 * 10^-6
Size 10000 Case 2	6.21 * 10^-2	0.000534
Size 10000 Case 3	1.25 * 10^-1	0.000323
Size 10000 Case 4	7.83 * 10^-3	0.000190
Size 100000 Case 1	3.91 * 10^-4	1.361
Size 100000 Case 2	6.22 * 10^0	0.0138
Size 100000 Case 3	1.24 * 10^1	0.00128
Size 100000 Case 4	7.68 * 10^-1	0.0055
Size 1000000 Case 1	3.90 * 10^-3	2.188*10^-5
Size 1000000 Case 2	X	X
Size 1000000 Case 3	X	X
Size 1000000 Case 4	7.68 * 10^1	0.2087

Table 4 - Merge Sort Results

Case	Average Time (seconds)	Standard Deviation
Size 10 Case 1	2.51 * 10^-6	6.43 * 10^-7
Size 10 Case 2	2.81 * 10^-6	4.42 * 10^-7
Size 10 Case 3	2.46 * 10^-6	5.58 * 10^-7
Size 10 Case 4	2.52 * 10^-6	5.59 * 10^-7
Size 1000 Case 1	2.13 * 10^-4	3.77 * 10^-5
Size 1000 Case 2	0.00024	1.95 * 10^-5
Size 1000 Case 3	0.00015	1.49 * 10^-5
Size 1000 Case 4	0.00013	1.4 * 10^-5
Size 10000 Case 1	0.0010	0.00011
Size 10000 Case 2	0.0014	5.65 * 10^-5
Size 10000 Case 3	0.0010	7.98 * 10^-5
Size 10000 Case 4	0.0010	3.82 * 10^-5
Size 100000 Case 1	0.0065	0.0016
Size 100000 Case 2	0.0160	0.000127
Size 100000 Case 3	0.0120	0.0007
Size 100000 Case 4	0.0121	9.07 * 10^-5
Size 1000000 Case 1	0.134	0.0077
Size 1000000 Case 2	0.180	0.00963
Size 1000000 Case 3	0.135	0.0007
Size 1000000 Case 4	0.139	0.0016

Table 5 - Quick Sort Results

Case	Average Time (seconds)	Standard Deviation
Size 10 Case 1	1.17 * 10^-6	2.01 * 10^-6
Size 10 Case 2	1.37 * 10^-6	3.09 * 10^-6
Size 10 Case 3	6.70 * 10^-7	4.72 * 10^-7
Size 10 Case 4	1.12 * 10^-6	1.95 * 10^-6
Size 1000 Case 1	3.13 * 10^-3	0.0003
Size 1000 Case 2	1.22 * 10^-4	7.14 * 10^-6
Size 1000 Case 3	9.5 * 10^-4	8.25 * 10^-5
Size 1000 Case 4	1.64 * 10^-4	5.73 * 10^-5
Size 10000 Case 1	3.09 * 10^-1	3.09 * 10^-1
Size 10000 Case 2	4.10 * 10^-3	4.09 * 10^-5
Size 10000 Case 3	8.34 * 10^-2	0.0135
Size 10000 Case 4	4.44 * 10^-3	0.00047
Size 100000 Case 1	3.11 * 10^1	3.1066
Size 100000 Case 2	3.22 * 10^-1	0.000424
Size 100000 Case 3	7.27 * 10^0	3.5893
Size 100000 Case 4	3.27 * 10^-1	0.0055
Size 1000000 Case 1	X	X
Size 1000000 Case 2	X	X
Size 1000000 Case 3	X	X
Size 1000000 Case 4	X	X

Discussion

In a broad sense, the results from the tests were as to be expected. Bubble and insertion sort were expected to run in n^2 time in most cases with the exception of the already sorted array being sorted in n time. The 10 percent shuffled array also did slightly better but still not as well as the fully sorted array as seen in tables 1 and 3. The selection sort also performed with the expected n^2 time in every case. You can see this in table 2 because every test for a single size will be raised to the same power signifying that the complexity for each test is approximately the same. Merge sort also exhibited this behavior as seen in table 4. This is to be expected because merge sort is in θ(n log n). The final test represented in the quick sort in table 5. Time complexity for quick sort is n^2 in the worst case and $nlogn$ in the best case. The end time complexity of the sort is determined on the pivot element that is chosen for the sort which is why there is variation between tests for how the algorithm performs. From the data set it is clear that both the merge sort and the quick sort are better choices than the three more basic algorithms. However, which algorithm is better between the merge and the quick sort is depended on the requirements of the larger program using the sort. Merge sort is better from a time complexity standpoint because it is guaranteed to run in $nlogn$ time but the space complexity can range from O(n) to O(n) + O(log n) because of the extra space needed for the sub arrays that are created. Conversely, the quick sort is not always as efficent from a time perspective – although it still runs in an average of $nlogn$ time – but since it sorts the array in-place it has a space complexity of O(1) (constant space complexity). Therefore, if time is the most valuable resource then it is best to use merge sort and if space is the most valuable resource then it is best to use quick sort.

Statistical Analysis

As discussed in the previous section, the means for each algorithm was more or less within the order of magnitudes that were expected from the theoretical analysis. For selection and bubble sort the general trend for the standard deviation seemed to be that it grew as the size of the array grew. This means that as the array got bigger the data points tended to fall further from the average. The insertion sort seemed to be more erratic than the selection and the bubble sort, although in general the standard deviation still grew with the size of the array. Mergesort seems to follow the same pattern of insertion sort but the quick sort is considerably more erratic.

Conclusion

In conclusion the best sorting algorithms were the quick sort and the merge sort. Each algorithm performed slightly differently and throughout the different tests that we ran on them. This project was also an excellent oppertunity to improve knowledge of object oriented design.