**Project 1**

**Benchmark of Sorting Algorithms**

Jheovanny Camacho

109697217

CSCI 115 – 35462

March 19, 2018

Introduction

This project will test five different sorting algorithms with four different cases, each with one of five different sizes. The goal is to examine the different times it takes for each algorithm to completely sort the given arrays. We will explore the time complexity of each result as well as the mean and standard deviation after testing each case one hundred times.

The five sorting algorithms consists of: Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort. The different cases of arrays to be sorted are: already sorted (best case scenario), reversed (worst case scenario), all elements are shuffled (random values), and ten percent of elements are shuffled. The different sizes are 10, 1000, 10000, 100000, 1000000.

Methods

The files used in this project are the header (Array.h) and implementation file (Array.cpp) of the template-using Array class, and the main file (CSCI 115 Project 1.cpp) which runs the main function. Every sorting algorithm has its own function, and any additional functions required, in the main file. It uses instances of the Array class, which contains functions that help create the different scenarios of arrays, as well as functions to display or swap elements in the given array. Essentially, a list of if-then statements is used after the user input, which determines which specific scenario is to be run. The majority of the main function is wrapped in a loop to allow for easier repeated tests, however a few lines of code (now commented out) were used previously for making sure each sorting algorithm was working properly. Mean and Standard Deviation, along with Variance, are calculated outside of the main loop. QueryPerformance functions were used to determine the elapsed time.

The following results in the two tables include each of the sorting algorithms, either with the mean/average or the standard deviation after testing each condition 100 times. Due to time constraints, exceptions have been made for the two largest sized arrays.\*\*

Results

Key: **S** = Sorted Array, **R** = Reversed Array, **SH** = Completely shuffled Array, **10** = Array that is sorted then shuffled at 10%, **\*** = too long to calculate in a suitable amount of time

*Note: All times are in microseconds (µs);*

*for any box colored in light orange, program would ultimately crash on that particular case*

This is a table of all the **Mean** times of every sorting algorithm being run 100 times, except the two largest cases.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Mean Times*** | | | | | | | | | | |
|  | 10 | | 1,000 | | 10,000 | | 100,000 (10 tests) | | 1,000,000 (3 tests) | |
| **Selection** | S | 0.02 | S | 1502.16 | S | 141751.16 | S | 13896356.70 | S | 1442953620.00 |
| R | 0.10 | R | 1656.84 | R | 151218.84 | R | 14420107.60 | R | \* |
| SH | 0.40 | SH | 1555.95 | SH | 144826.15 | SH | 13872506.10 | SH | \* |
| 10 | 0.16 | 10 | 1533.03 | 10 | 140065.33 | 10 | 13846327.50 | 10 | 1429708821.22 |
| **Insertion** | S | <0.01 | S | 3.90 | S | 51.13 | S | 446.47 | S | 4689.96 |
| R | 0.52 | R | 5803.65 | R | 416280.66 | R | 32232687.30 | R | \* |
| SH | 0.19 | SH | 4006.82 | SH | 214682.28 | SH | 16075061.70 | SH | \* |
| 10 | 0.04 | 10 | 29.07 | 10 | 4511.95 | 10 | 209083.95 | 10 | 17336976.66 |
| **Bubble** | S | 0.03 | S | 4301.08 | S | 329085.86 | S | 25388389.50 | S | 2625645668.00 |
| R | 0.64 | R | 8267.47 | R | 481677.78 | R | 47009515.60 | R | \* |
| SH | 0.77 | SH | 6730.01 | SH | 583009.25 | SH | 52753596.80 | SH | \* |
| 10 | 0.05 | 10 | 6416.92 | 10 | 326441.90 | 10 | 25324280.30 | 10 | \* |
| **Merge** | S | 0.77 | S | 89.08 | S | 1183.17 | S | 11823.47 | S |  |
| R | 0.70 | R | 122.35 | R | 1176.69 | R | 12072.53 | R |  |
| SH | 1.24 | SH | 172.79 | SH | 1575.70 | SH | 17111.19 | SH |  |
| 10 | 0.79 | 10 | 115.78 | 10 | 1169.32 | 10 | 12452.61 | 10 |  |
| **Quick** | S | 0.88 | S | 3926.48 | S | 343703.22 | S |  | S |  |
| R | 0.73 | R | 2683.04 | R | 243233.64 | R |  | R |  |
| SH | 0.48 | SH | 178.70 | SH | 4968.92 | SH | 365468.73 | SH | 36846258.00 |
| 10 | 0.67 | 10 | 3984.65 | 10 | 335115.51 | 10 |  | 10 |  |

This is a table of the **Standard Deviations** of all cases:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Standard Deviation*** | | | | | | | | | | |
|  | 10 | | 1,000 | | 10,000 | | 100,000 (10 tests) | | 1,000,000 (3 tests) | |
| **Selection** | S | 0.14 | S | 69.18 | S | 1188.30 | S | 101518.28 | S | 33405879.10 |
| R | 0.41 | R | 136.35 | R | 7495.16 | R | 19915.83 | R | \* |
| SH | 0.49 | SH | 71.76 | SH | 5100.03 | SH | 23155.69 | SH | \* |
| 10 | 0.42 | 10 | 91.95 | 10 | 1200.85 | 10 | 12792.30 | 10 | 44905739.84 |
| **Insertion** | S | <0.01 | S | 0.88 | S | 19.40 | S | 124.78 | S | 1360.82 |
| R | 0.50 | R | 1555.08 | R | 45063.69 | R | 141382.70 | R | \* |
| SH | 0.56 | SH | 1577.00 | SH | 31725.43 | SH | 55336.31 | SH | \* |
| 10 | 0.28 | 10 | 10.98 | 10 | 1889.27 | 10 | 28905.97 | 10 | 632043.14 |
| **Bubble** | S | 0.22 | S | 1046.88 | S | 41676.38 | S | 221432.76 | S | 4294970625.46 |
| R | 0.56 | R | 2004.08 | R | 15336.92 | R | 255156.02 | R | \* |
| SH | 0.68 | SH | 1774.52 | SH | 62083.17 | SH | 157988.62 | SH | \* |
| 10 | 0.26 | 10 | 4342.31 | 10 | 36494.42 | 10 | 167727.24 | 10 | \* |
| **Merge** | S | 1.65 | S | 16.48 | S | 123.13 | S | 481.65 | S |  |
| R | 2.47 | R | 22.37 | R | 127.50 | R | 745.29 | R |  |
| SH | 2.15 | SH | 30.12 | SH | 147.49 | SH | 682.32 | SH |  |
| 10 | 1.68 | 10 | 27.41 | 10 | 119.71 | 10 | 765.23 | 10 |  |
| **Quick** | S | 0.47 | S | 485.13 | S | 17170.95 | S |  | S |  |
| R | 2.73 | R | 294.44 | R | 13145.97 | R |  | R |  |
| SH | 1.66 | SH | 33.13 | SH | 1062.82 | SH | 39313.98 | SH | 811727.25 |
| 10 | 2.06 | 10 | 807.15 | 10 | 3191.18 | 10 |  | 10 |  |

\*\* For 100000, sorting an array may take a little over 20 minutes to do 100 tests on a single case, so only 10 tests will be made per case

* + otherwise: 23 min \* 4 scenarios \* 5 sorting algorithms = 7.6 hours total

For 1000000, sorting an array may take a little over 20 minutes to do *1 test* on a single case, so only 3 tests will be made per case

* + otherwise: 23 min \* 100 tests \* 4 \* 5 = 766 hours total

Statistical Analysis: Interpreting the Results

According to the results, the overall fastest and most efficient sorting algorithm out of all five is: **Merge Sort**.

Selection Sort had the most consistent times, very fast when the array was small and very slow when the array was large, in all cases. For Insertion Sort, processing an array that is already sorted was the fastest among that case for all sizes and algorithms, competing with Merge Sort. However, when it came to the reversed and shuffled cases, Insertion Sort took as long as Selection Sort, sometimes even longer. In fact, Insertion Sort had the biggest struggle among reversed arrays. Bubble Sort is about on par with Insertion Sort, except lacking the speedy “already sorted” case advantage.

Merge Sort overall was dramatically faster in all cases in any size among all five of the sorting algorithms. It did not take a hundred times longer to sort even when the array was ten times larger than the previous. Quick Sort took a very long time when sorting the already sorted array. Sorting times when the array was shuffled or reversed would be less. Unfortunately, some issues with segmentation faults had caused the program to crash and be unable to complete Merge Sort or Quick Sort on the largest array. However, given the partial results, it can be clear that Merge Sort will still be the winner in that case as well.

Theoretical Analysis vs. Times Observed

When taking into consideration the time complexity: the first three sorting algorithms, Selection, Insertion, Bubble, are inferior as their average case is O(n2). Merge and Quick Sort have a better time complexity as their average case is O(n\*log(n)).

Looking closely at Selection Sort, we can see that it follows its comparison formula which states: N\*(N-1)/2 where N is the number of items in the array. The result is the number of comparisons that is required to sort an array. Insertion Sort has a less comparison demanding formula: N\*(N-1)/4 and this is due to each passing only needing to compare, on average, half of the maximum number of items. Bubble Sort also has a comparison formula of N\*(N-1)/2. Both Insertion and Bubble Sort will only have a comparison formula of (N-1) when the array is already sorted, and we see how well Insertion Sort performed on that particular scenario. Bubble Sort, however, performed far below my expectations. With these formulas and the time complexities, we can see how times grew exponentially as predicted.

Both Merge and Quick Sort have a different approach to sorting: they rely on recursion and an additional function to support their “Divide and Conquer” method of sorting. Taking a look at Merge Sort, we immediately see a dramatic difference in mean times compared to the previous three sorting algorithms. Rather than the mean times growing exponentially as the array grew larger, they grow gradually and stay within close proximity, in a logarithmic fashion as the time complexity suggests. Quick Sort, which is supposedly the fastest among all five algorithms, seemed to underperform. Its greatest strength was sorting completely shuffled arrays; however, other scenarios had times that were comparable with the first three. Given what we know, when an array is already sorted, Quick Sort can be one of the worst algorithms, and it definitely shows. Its worst case is O(n2), just like the first three.

Conclusion

After studying all the recorded data and reviewing the theoretical analysis, I can see that these numbers, for the most part, follow what was predicted. One of the two algorithms with superior time complexity than the rest had proven to be faster and more efficient. We see patterns that have also been predicted by complexity and the way each algorithm does their comparisons. In the end, each algorithm has their own weaknesses and strengths. It really all depends on the specific scenario.