PA2 Report

By

William Box

CSI 281
Data Structures & Algorithms
Fall 2022-23

1. Introduction

For this experiment, I will be empirically testing all 6 of the sorting algorithms we learned in class in order to determine which algorithm is best under which conditions.

2. Background

Provide information on each of the algorithms on their characteristics, advantages and disadvantages. Also, provide information on their performance analysis.

For this experiment the Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Merge Sort, and Quick Sort algorithms were all tested. As a general rule, the algorithms will be listed in order of slowest to fastest, with the second generalization being that the faster an algorithm runs the more memory it will utilize. It would also happen that the "faster" algorithms are not necessarily faster, but rather are more likely to run at nearly the same speed no matter how unorganized the data set is. From here on, this will be referred to as the "stability" of an algorithm.

As a summary, the algorithms should have the following behaviors

Bubble Sort
- Easy to implement
- Low memory utilization
- Unstable performance

Insertion Sort
- Low memory utilization
- Minimizes swaps performed, resulting in faster performance
- Unstable performance

Selection Sort
- Low memory utilization

Shell Sort
- Low memory utilization
- Stable performance

Merge Sort
- Highly stable performance
- High speed
- Highest memory utilization

Quick Sort
- Stable performance
- High speed
- High memory utilization

3. Implementation Detail

The algorithms were all tested using the same 3 automatically generated data sets. These data sets were then pre-sorted and arranged into three separate states, random, sorted, and reversed (average, worst, and best cases), giving us a total of 9 different files. From there, each algorithm was tested 3 times with each file, and asked to sort 100, 10000, and 750000 numbers from each of the 9 files.

The algorithms were all written as templated functions, capable of performing a sort on a static or dynamic array of any data type capable of using the comparison operator.

4. Experimentation Detail

The experiment was ran on a Windows Surface Book 3 with the following specs

a. 32 GB Total physical memory, 19.9 GB Available physical memory
b. Intel Core i7-1065G7, 4 core processor
c. 1.30 GHz
d. System type: 64 bits

Summary Data

Algorithm: Bubble Sort			
N	**Dataset #1**[1]	**Dataset #2**	**Dataset #3**
100	2.12e-05	3.00e-07	2.79e-05
10,000	2.01e-01	1.38e-05	2.26e-01
750,000	1.73e 03	1.04e-03	1.30e 03

Algorithm: Insertion Sort			
N	**Dataset #1**[2]	**Dataset #2**	**Dataset #3**
100	5.9e-06	4.67e-07	1.03e-05
10,000	4.59e-02	2.33e-05	8.66e-02
750,000	2.69e 02	1.98e-03	5.19e 02

[1] Dataset #1 is average case, dataset #2 is best case and dataset #3 is worst case.
[2] Dataset #1 is average case, dataset #2 is best case and dataset #3 is worst case.

Algorithm: Selection Sort			
N	**Dataset #1**[3]	**Dataset #2**	**Dataset #3**
100	1.03e-05	8.10e-06	1.23e-05
10,000	7.12e-02	6.64e-02	9.01e-02
750,000	3.97e 02	3.89e 02	8.94e 02

Algorithm: Shell Sort			
N	**Dataset #1**[4]	**Dataset #2**	**Dataset #3**
100	2.15e-05	4.67e-07	2.36e-05
10,000	2.01e-02	1.40e-05	2.18e-01
750,000	1.70e 03	1.41e-03	1.34e 03

Algorithm: Merge Sort			
N	**Dataset #1**[5]	**Dataset #2**	**Dataset #3**
100	5.51e-05	5.35e-05	5.31e-05
10,000	6.78e-03	5.69e-03	6.92e-03
750,000	4.97e-01	4.37e-01	4.34e-01

Algorithm: Quick Sort			
N	**Dataset #1**[6]	**Dataset #2**	**Dataset #3**
100	6.20e-06	3.33e-06	3.97e-06
10,000	9.67e-04	4.38e-04	3.10e-04
750,000	9.47e-02	2.67e-02	2.71e-02

[3] Dataset #1 is average case, dataset #2 is best case and dataset #3 is worst case.
[4] Dataset #1 is average case, dataset #2 is best case and dataset #3 is worst case.
[5] Dataset #1 is average case, dataset #2 is best case and dataset #3 is worst case.
[6] Dataset #1 is average case, dataset #2 is best case and dataset #3 is worst case.

5. Discussion and Conclusion

As seen with the data summarized above, the algorithms performed as expected, with the algorithms near the beginning of the list performing worse overall than the ones later in the list.

However, there are some notable behaviors found when examining the results of smaller data sets. When one looks for the overall fastest average case with the 100 int data set, Quick Sort is the winner, but it only just barely beats out Insertion Sort, which is less than 10% slower. In addition, one will see that in the best case scenarios, Bubble Sort is actually an order of magnitude faster than Quick Sort or Merge Sort, and is the fastest of all the algorithms tested. In fact, of all the algorithms tested, Bubble Sort had the most unstable performance, with the method running dramatically faster in the best case scenario than other “faster” algorithms.

When examining the effects of the data set used, I believe the contrast between Quick Sort and Bubble Sort gives us the most information to look at. With a “stable” sorting algorithm, the time it takes to sort a given data set only really varies based on the size of the data set. As such, Quick Sort takes roughly the same amount of time to sort the average, best, and worst case scenarios of each sample size. Bubble Sort on the other hand wildly varies in performance between the average, best, and worst case scenarios, with the difference becoming more extreme the larger the data set is.

This tells us two things, first, that stable algorithms are nice if you have no idea what the incoming data will look like, as they will not vary much in performance. But unstable algorithms like Bubble Sort can actually be faster than the “fast” algorithms when the data is nearly sorted, and the performance difference when examining small data sets is so minimal as to make the extra effort spent on Quick Sort simply not worth it.

When looking at general performance however, the algorithms can be listed in this order, from fastest to slowest.

- Quick Sort
- Merge Sort
- Insertion Sort
- Selection Sort
- Shell Sort
- Bubble Sort

As for my own personal preference, I like the Quick Sort algorithm. Its method is intuitive to my senses, and its performance is highly stable, making it useful for most scenarios. However, I do believe that there are better alternatives for specific edge cases.

6. References

N/A

7. Appendix

You must have one table for each algorithm to report on the different run-time requirements.

Algorithm: Bubble Sort			**Dataset #:** Average Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	2.30e-5	2.08e-05	1.97e-05	2.12e-05
10,000	2.01e-01	2.04e-01	1.98e-01	2.01e-01
750,000	1.73e 03	1.72e 03	1.74e 03	1.73e 03

Algorithm: Bubble Sort			**Dataset #:** Best Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	3.00e-07	3.00e-07	3.00e-07	3.00e-07
10,000	1.41e-05	1.34e-05	1.38e-05	1.38e-05
750,000	1.14e-03	1.04e-03	9.34e-04	1.04e-03

Algorithm: Bubble Sort			**Dataset #:** Worst Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	3.05e-05	3.00e-05	2.31e-05	2.79e-05
10,000	2.19e-01	2.31e-01	2.31e-01	2.26e-01
750,000	1.29e 03	1.29e 03	1.31e 03	1.30e 03

Algorithm: Insertion Sort			**Dataset #:** Average Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	5.8e-06	5.3e-06	6.6e-06	5.9e-06
10,000	4.78e-02	4.53e-02	4.46e-02	4.59e-02
750,000	2.89e 02	2.59e 02	2.58e 02	2.69e 02

Algorithm: Insertion Sort			**Dataset #:** Best Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	5.00e-07	5.00e-07	4.00e-07	4.67e-07
10,000	2.31e-05	2.33e-05	2.36e-05	2.33e-05
750,000	1.80e-03	2.20e-03	1.92e-03	1.98e-03

Algorithm: Insertion Sort			**Dataset #:** Worst Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	8.9e-06	1.1e-05	1.1e-05	1.03e-05
10,000	8.77e-02	8.35e-02	8.85e-02	8.66e-02
750,000	5.26e 02	5.15e 02	5.15e 02	5.19e 02

Algorithm: Selection Sort			**Dataset #:** Average Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	1.12e-05	9.90e-06	9.80e-06	1.03e-05
10,000	7.09e-02	6.79e-02	7.47e-02	7.12e-02
750,000	3.99e 02	4.00e 02	3.91e 02	3.97e 02

Algorithm: Selection Sort			**Dataset #:** Best Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	7.7e-06	7.6e-06	9e-06	8.10e-06
10,000	6.78e-02	6.63e-02	6.52e-02	6.64e-02
750,000	3.88e 02	3.88e 02	3.92e 02	3.89e 02

Algorithm: Selection Sort			**Dataset #:** Worst Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	1.3e-05	1.22e-05	1.17e-05	1.23e-05
10,000	0.0821348	0.0935899	0.094473	9.01e-02
750,000	8.93e 02	8.90e 02	8.98e 02	8.94e 02

Algorithm: Shell Sort			**Dataset #:** Average Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	2.32e-05	2.09e-05	2.04e-05	2.15e-05
10,000	2.02e-02	2.04e-02	1.96e-02	2.01e-02
750,000	1.72e 03	1.71e 03	1.67e 03	1.70e 03

Algorithm: Shell Sort			**Dataset #:** Best Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	4.00e-07	6.00e-07	4.00e-07	4.67e-07
10,000	1.45e-05	1.42e-05	1.34e-05	1.40e-05
750,000	9.75e-04	1.54e-03	1.72e-03	1.41e-03

Algorithm: Shell Sort			**Dataset #:** Worst Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	2.41e-05	2.33e-05	2.33e-05	2.36e-05
10,000	2.21e-01	2.17e-01	2.17e-01	2.18e-01
750,000	1.36e 03	1.35e 03	1.30e 03	1.34e 03

Algorithm: Shell Sort			**Dataset #:** Worst Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	2.41e-05	2.33e-05	2.33e-05	2.36e-05
10,000	2.21e-01	2.17e-01	2.17e-01	2.18e-01
750,000	1.36e 03	1.35e 03	1.30e 03	1.34e 03

Algorithm: Merge Sort			**Dataset #:** Average Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	5.79e-05	5.42e-05	5.32e-05	5.51e-05
10,000	0.0065709	0.0068641	0.0069039	6.78e-03
750,000	0.498908	0.49214	0.501595	4.97e-01

Algorithm: Merge Sort			**Dataset #:** Best Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	5.65e-05	5.31e-05	5.1e-05	5.35e-05
10,000	0.0058403	0.0055363	0.0056896	5.69e-03
750,000	0.441933	0.436646	0.431561	4.37e-01

Algorithm: Merge Sort			**Dataset #:** Worst Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	5.73e-05	5.2e-05	5.01e-05	5.31e-05
10,000	0.0079332	0.005762	0.0070572	6.92e-03
750,000	0.43485	0.430686	0.437628	4.34e-01

Algorithm: Quick Sort			**Dataset #:** Average Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	6.50e-06	6.30e-06	5.80e-06	6.20e-06
10,000	9.12e-04	9.60e-04	1.03e-03	9.67e-04
750,000	0.0929093	0.0970355	0.0941101	9.47e-02

Algorithm: Quick Sort			**Dataset #:** Best Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	in 3.5e-06	3.1e-06	3.4e-06	3.33e-06
10,000	0.0005981	0.0003303	0.0003847	4.38e-04
750,000	0.0272625	0.0271201	0.0256528	2.67e-02

Algorithm: Quick Sort			**Dataset #:** Worst Case	
N	**Run #1**	**Run #2**	**Run #3**	**Average**
100	3.9e-06	3.8e-06	4.2e-06	3.97e-06
10,000	0.0003309	0.0002777	0.0003229	3.10e-04
750,000	0.02835	0.0263086	0.0266046	2.71e-02