Profiling Research Paper

Analysis of Sorting Algorithms Using a Code Execution Profiler

Course: CSCI 2210-002, Spring 2019

Author: Ryan Shupe

Table of Contents

[Abstract 2](#_Toc4252243)

[Introduction 3](#_Toc4252244)

[The Problem 3](#_Toc4252245)

[The Experiment 4](#_Toc4252246)

[Results 5](#_Toc4252247)

[Conclusion 7](#_Toc4252248)

[Appendix 1 - Data 8](#_Toc4252249)

[Appendix 2 – Graphs 9](#_Toc4252250)

[References 12](#_Toc4252251)

# Abstract

An experiment was conducted to research and gather data from a set of different sorting algorithms. The goal was to see if one algorithm performs all around better than the other ones and if there is a best algorithm depending on the set of data given, small, large, random, sorted, etc. The results showed that the Quick Sort Median of 3 sorting algorithms performed all-around best with small and large data in the scope of the experiment. The results pointed out the efficiency of the Counting Sort with large amounts of data, and showed algorithms that should be avoided in almost all situations. The Original Quick Sort performed the worst in all cases of the experiment and is the clear most inefficient. If your data is already sorted than Sink Sorting performs the most admirable. Any deviation from being sorted, it becomes more inefficient. One of the trials in the experiment was to sort the data what was given in a reverse order, and the Sink Sorting performed the worst by a large margin.

# Introduction

Computers are really fast. It is not only because they can do many small tasks very quickly, it is because we also write well and efficient code. We develop the most efficient algorithm for the situation at hand which will make our program run faster with fewer performance issues. In this research paper, I will explain an experiment that I used to analyze how different methods of sorting numbers affects computer performance. I will show graphs and the raw data to explain my points and reasoning on why a particular algorithm runs more efficiently, or possibly multiple algorithms that are good in any situation. The expectation is that different algorithms will outperform each other depending on the data.

# The Problem

The problem is to investigate the nature of how an algorithm reacts in various circumstances. Is there a best algorithm for all possible situations? The best way to analyze how different algorithms work and perform is to conduct an experiment to gather the execution data and compare and contrast.

# The Experiment

I have written a simple program to test various sorting algorithms. The program fills a list of N integers where N is going to change to create a larger data size and calls a method to sort the data. N started at an initial value of 100, then 1000, and 10000, to give an accurate representation of how a particular sorting algorithm reacts to larger sets of data. The sorting algorithms that I tested are, Sink Sorting, Insertion Sorting, Merge Sorting, Quick Sort, Median of Three Quick Sorting, Shell Sorting, Count Sorting, and Radix base 10 Sorting. I profiled the execution using Microsoft Visual Studio 2017’s execution profiler using instrumentation.

After each trial of running the profiler, I gathered the total execution run time of the program 3 times for each sorting algorithm and the value of N to give an accurate representation about how efficient each algorithm takes to run. The different types of data changed after complete runs from being all random, already sorted, reverse sorted order, and 10% random data. After gathering all the data, I created graphs to visually show the results and to be able to compare and contrast and see which algorithms work best in a wide range of circumstances.

# Results

First I conducted the trials of when N = 100 for all the types of sorting methods and all types of data. Appendix 1, Table A shows all of the data I gathered from the total runtime execution of the program. By a quick glance, we can determine that there are some that seem more efficient. By visually graphing the data (Appendix 2, Graph E) we can see that no matter what kind of data we have, Shell Sorting seems to be the most efficient by a relatively small margin. All of the algorithms seemed to have run at the same efficiency with the exception of Radix and Counting Sort.

When increasing the value of N to 1000, we can see which algorithms are more efficient than others (Appendix 1, Table B). In Appendix 2, Graph F, we can see that 5 algorithms perform a lot better than the remaining ones. Original Quick is not shown on the graph due to the fact it took a very substantial amount of time to load the results. This will start to become more common over many types of algorithms as we increase the value of N and change the types of data. Quick Median of 3 sorting performs the best out of all types of data here. It performed the 4th best when N was equal to 100. We can already see from the graphs that the Median of 3 methods of quick sorting is exponentially more efficient than the original quick sorting it derived from.

Increasing the value to 10000, we can see slightly similar results from when N was equal to 1000 (Appendix 1, Table C). This time over the course of all the different types of data, we can see that Counting sort performs the best with the Quick Median of 3 Sorting in a close 2nd. Many different types of data were too slow to accurately gather when N is set to a very large number. Over half of the sorting algorithms I tested are shown as being very inefficient compared to Counting, Quick median of 3, Shell Sort, and Radix.

With looking at the data from all different types of data, and whichever value of N there is, it seems that the Quick Median of 3 sorting is the best all-around sorting method you can use. With increasing the number of N, the Quick median of 3 sorting algorithm is consistent at the beginning of the graphs showing that the algorithm is very efficient. Another thing that is consistent across the data is how very inefficient the Original Quick Sort is. When N is equal to 1000 we see that it is at the end of the graph and doesn’t have a bar, the same thing occurs when N is equal to 10000. This makes sense because the big-oh notation for Quick Sort is O(n^2) and according to the performance, it confirms the theoretical analysis of the particular algorithm. As you increase the amount of data, the longer the amount of time it is going to take to run. This same principle applies to most of the algorithms at the far end of the graph in Appendix 2, Graph’s E and F.

Looking at the results from each data set separately we can see a similar story. When the data is 100% random, the efficiency of Quick Median of 3 goes up as N increases and the Original Quick Sort becomes more inefficient. Counting Sort also seems to be more efficient than the Quick Median of 3 when N is exponentially large. The big oh Notation for Counting Sort is O(N+K). The data shows that Counting Sort seems to be extremely efficient if you think about it in a perspective outside of this conducted experiment for a completely random set of data (Appendix 2, Graph A).

If we look at already sorted data, we see a different story. Sink sort performs the most efficient. (Appendix 2, Graph B) While Original Quick Sorting and Selection Sorting perform relatively the same as they have been, it makes sense why Sink Sort is the winner here. Sink Sorting compares each number to check if it is in order and goes down the data, there is nothing to sort because all the data is already in order. If a programmer wanted to sort already sorted data, then using Sink Sort would be best in this case.

Looking at the data that is sorted, but in a reverse order shows that Sink Sorting perfoms much worse. This is due to how Sink Sorting works. It is the most inefficient here as N gets large. The most efficient is Shell Sorting when N is less than 10000. When it is greater than 10000, we seethat Counting Sort more efficient, but it is not more effecient when N is a lower number. (Appendix 2, Graph C) In this set of graphs, we can see that the Quick Median of 3 Sorting is very efficient in all cases, performing the 2nd best in all of the graphs in the set.

If we look at the data where 10 % is random numbers and the rest is already sorted, it looks vaguely familiar to the graphs when the data was 100% random. (Appendix 2, Graph D) As N gets large, over half of the algorithms become very inefficient. The Quick Median of 3 once again shows that it is the all-around most efficient, while the Counting Sort became the best when N is exponentially large.

# Conclusion

As shown with the research and experimentation above, it is concluded that the most efficient sorting algorithm all-around is the Quick Sorting Median of 3. Throughout the experiment, the algorithm performed very efficiently in almost all recorded cases. If a programmer is sorting a relatively small amount of data, then using this algorithm is the best all-around case. However, if the data is already in a sorted form, then the Sink Sort is the most efficient. In the experiment, it was shown that as N increased to 10000 and above, the method of Counting Sort was the most efficient as the number of data increased. It was shown to be a little more efficient than the Quick Sorting Median of 3 in the graphs, but as the amount grew larger outside of the scope of the experiment, it showed to be more efficient.

Regarding inefficient algorithms, the main method of sorting I would advise staying away from is the Original Quick Sorting. This algorithm performed the worst in all of the trials, and substantially worse than the others that were shown to be inefficient as N increased. Even when N was a lower number it was on the higher end of the graphs, showing its inefficiency. The most interesting algorithm I studied was Counting sort. If we look at the data from when N was equal to 100 with all types of data, we can see that Counting Sort performed the worst. However, if we increase N its efficiency went up dramatically to eventually be the most efficient as N became exponentially large.

# Appendix 1 - Data

Table A

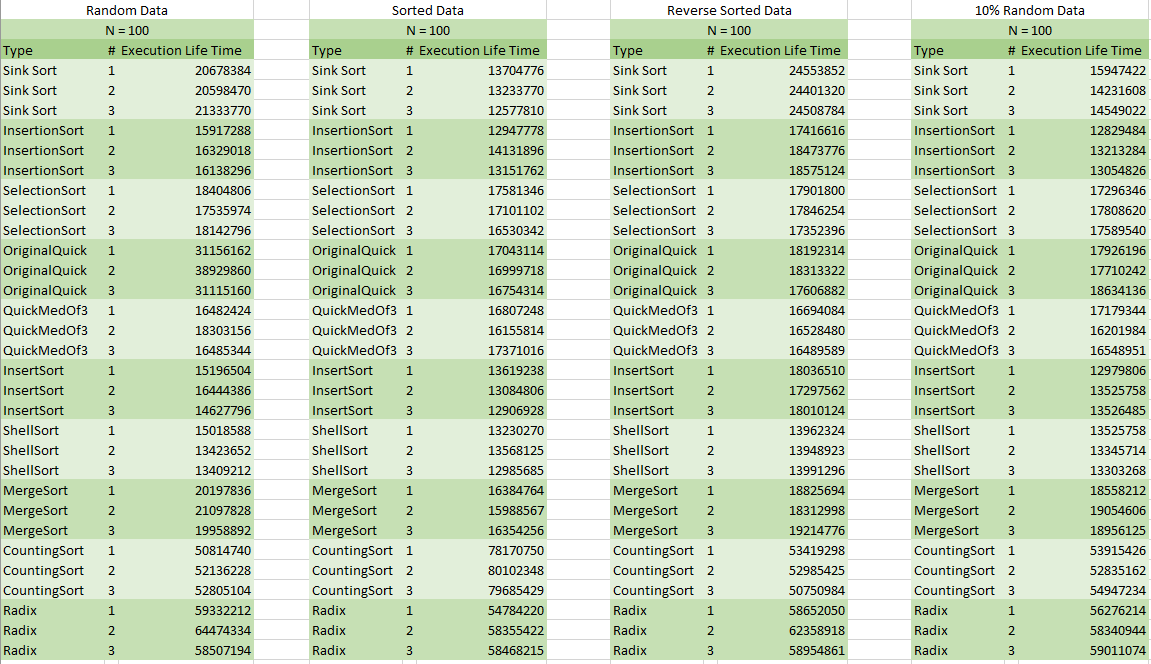


Table B

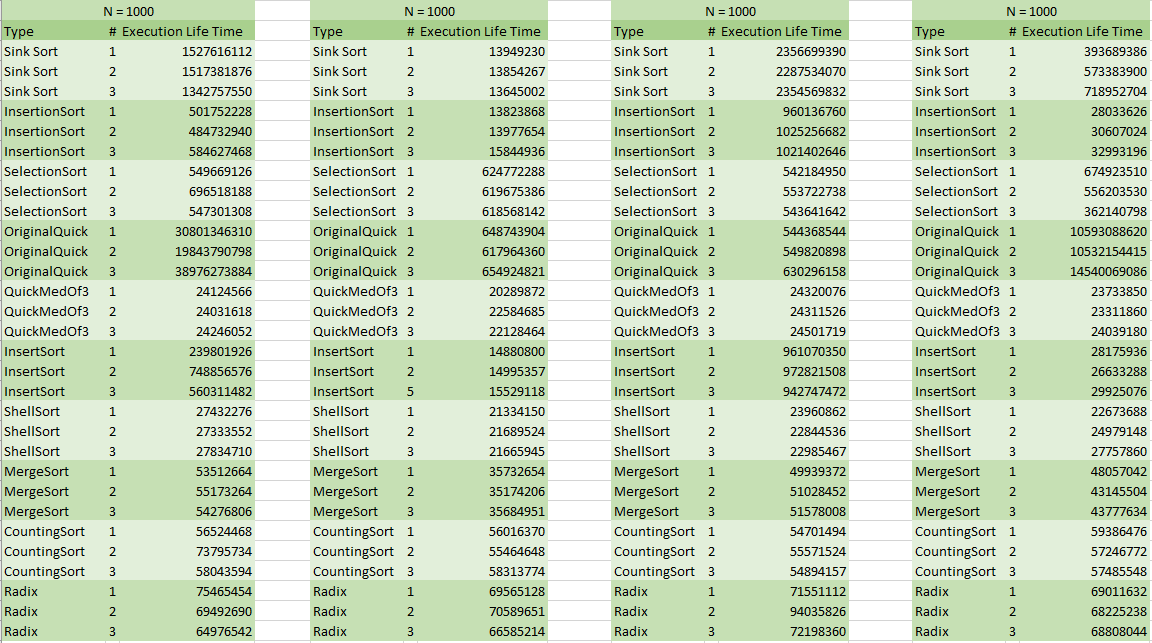
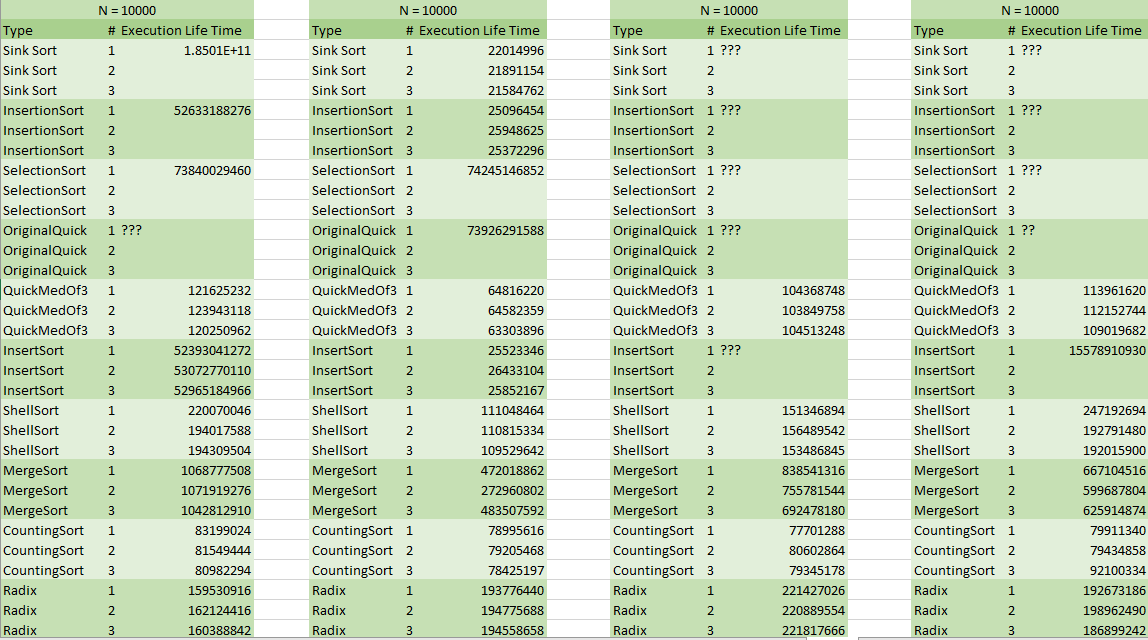
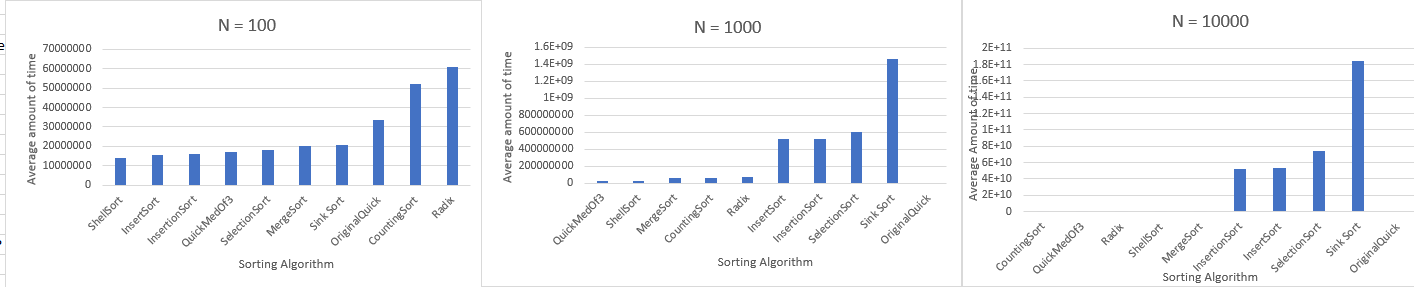


Table C

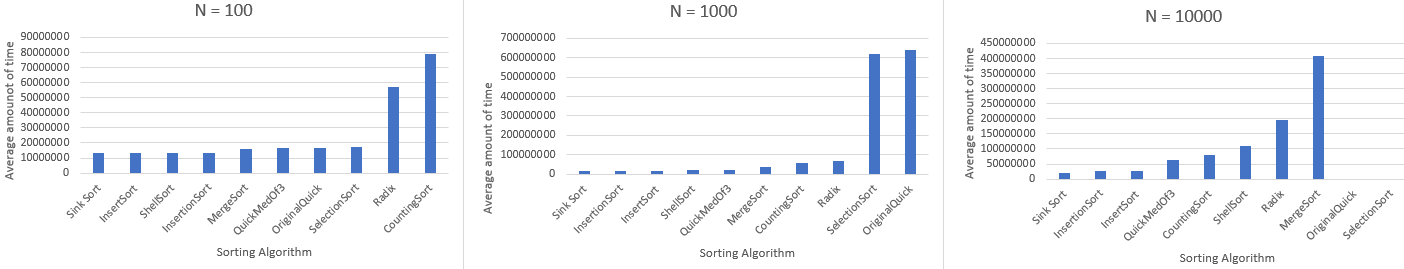


# Appendix 2 – Graphs

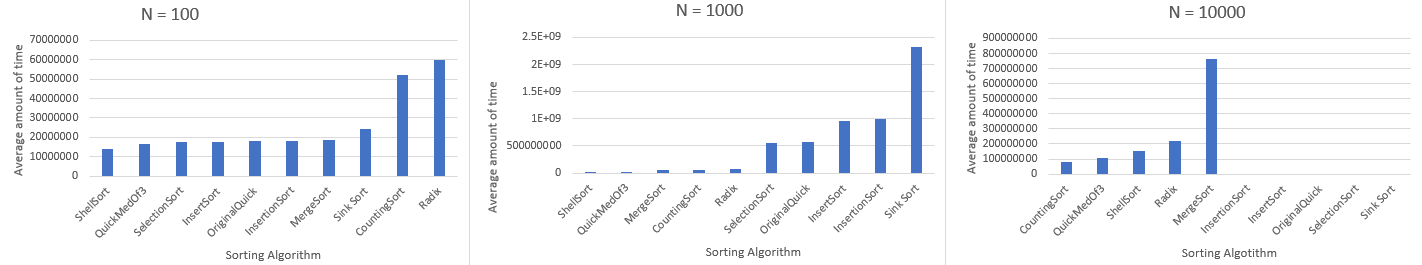
Graph A – Random data



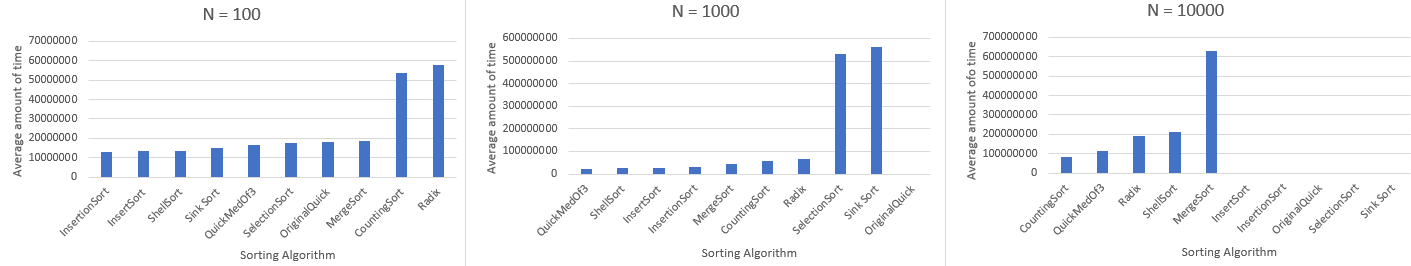
Graph B – Sorted Data



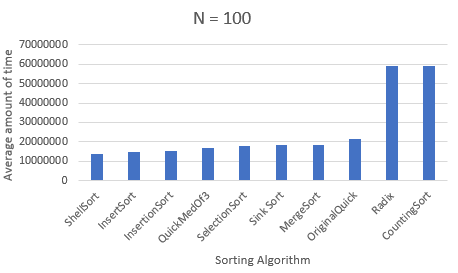
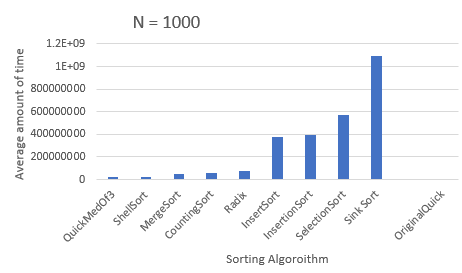
Graph C – Reverse Sorted Data



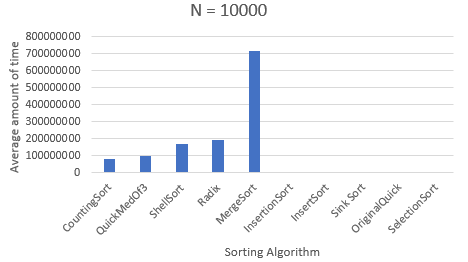
Graph D – 10% Random Data



Graph E – Average of all when N = 100 Graph F – Average of all when N = 1000

Graph G – Average of all when N = 10000



# References

*Complexity and Big-O Notation*, pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html.

De S Oliveira, Fabiano, and Valmir Barbosa. "Counting Independent Terms in Big-oh Notation." *ArXiv.org* (2015): ArXiv.org, Dec 31,

2015. Web.

“Know Thy Complexities!” *Big*, bigocheatsheet.com/.

“Sorting Algorithms.” *BetterExplained*, betterexplained.com/articles/sorting-algorithms/.