Profiling Research Paper

Analysis of Sorting Algorithms Using a Code Execution Profiler

Course: CSCI 2210-002, Fall 2021

Author: Calen Cummings

Table of Contents

[Introduction 2](#_Toc338687108)

[The Problem 2](#_Toc338687109)

[Solution Plan 2](#_Toc338687110)

[Tasks Completed 2](#_Toc338687111)

[Results and Conclusions 3](#_Toc338687112)

[Summary 3](#_Toc338687113)

[Appendix 1 – Data and Graphs 5](#_Toc338687114)

[References 9](#_Toc338687115)

# Introduction

In this report, we will be analyzing the varying performance of 9 different sorting algorithms applied to the same set of data. By altering the amount of items to sort, we can establish patterns such as the most efficient and well-rounded algorithm, the algorithm that performs the worst in most general cases, and which algorithms improve their performance with more/less available data to sort. Not only does this process reveal characteristics and performance of different algorithms, but by using the execution profiler to log how each algorithm performs against the rest, we can begin to understand how to use the tool and find additional inferences/meaning from our observed results.

# The Problem

We want to perform 9 different sorting algorithms on the same set of randomized data to determine the “best” sorting algorithm under varying values of N. Determining the best or most appropriate algorithm typically relies many factors, especially in concern of speed vs. accuracy. For our purposes, we want to identify the best algorithm in a given scenario as the one that demonstrates a consistent efficiency and typically occupies less of the overall execution time. The higher percentage of execution time that an algorithm takes up, the less efficient it is. As stated, efficiency/performance will vary with increasing numbers of N, our variable to represent the amount of random integers that need to be sorted within a list. Additionally, we want to observe how the algorithms perform on data that is not entirely random or is previously sorted to some extent. To achieve this, we will test completely random data against data that is already in order, data in reverse order, and mostly sorted data with about 10% remaining random. By introducing these extra qualifiers for the performance of our algorithms, we can observe how more factors affect each algorithm in positive and negative ways.

# Solution Plan

To begin testing our algorithms, we will start with a random list of 100 integers and fill that data into different lists for each of our algorithms to sort, so that we can get empirical evidence for each algorithm’s work on the data set. This process will be repeated with lists of 1,000 and 10,000 integers to test how increasing the amount of data to work on will impact the algorithm’s completion time. After each execution of the sorting methods on a set of data, we will profile the program to see just how long each algorithm is taking to process the data within the program’s runtime, and from this data we will collect observations and be able to form some patterns/conclusions.

# Tasks Completed

To start, we initialize the program with a list of N integers, filled to N count with random integers. The fully random list is then worked on by all 9 of our sorting algorithms, each performed on their own list of identical data so as to achieve the least bias possible. Initially, we do not set a limit on what random integers are allowed, as we have not run into any issue. After executing the program with completely random data, we profile the execution and log the data, specifically the Inclusive time in percentage and the inclusive time in milliseconds that the algorithm was in use in the program’s runtime. The inclusive percentage allows us to see how much time the algorithm is taking up relative to the others, and the inclusive time in milliseconds is an average across all of the calls of the method, but this average allows us to infer how efficient the algorithm is. Following the collection of data, we then test the algorithms on a set of data that is already in order, and record the profiling results. After a set of ordered data, we test a set in reverse order, and then finally a set that is 90% ordered and 10% random. To achieve the latter parameter, we need to sort only 90% of the data set before feeding it into the algorithms that we are testing. Once all four scenarios have been tested for N, we increase it. For this report, we started with an N value of 100 and increased to 1,000; once testing for N = 1,000 is completed, we increase N to 10,000. Increasing N revealed the importance of choosing the “best”/most appropriate algorithm for large data sets, as there was a tremendous gap in the time it took for the profiler to completely finish and display the report between N = 1,000 and N = 10,000. Early testing with N = 10,000 led to many crashes and OutOfMemory exceptions, so a limit is placed on the maximum random value that can be produced for the data set at 1,000,000, a large enough number to give us a wide sample of numbers to work with while also preventing situations where a Counting sort done on a data set with a max value of 2,109,325,391 tries to create an array with over two billion entries to get its sorting done.

# Results and Conclusions

From the testing, we find different results for each value of N. For N = 100, Counting Sort stands out as the worst or most inefficient sorting method tested, dominating each of the other algorithms in inlcusive percentage and boasting a lousy average of 3+ ms to perform each call in every tested scenario [1][2][3][4]. At the same N value, our testing did not identify an obvious best algorithm for every scenario, but Insertion sort performed the best on random and sorted data, but lagged behind in the reversed data and 10% random data scenarios, showing less efficiency as the data becomes more complex to order [1][2][3][4]. Because our N value leaves our data set so (comparatively) small, the execution time is largely filled by standard processes in the program’s execution, with none of our tested algorithms breaking over 11.4% of the execution’s runtime. The differences in performance with this N value are measurable, but are largely negligible in terms of which offers the fastest and best performance. For N = 1,000, we begin to see a clear loser in every scenario tested: Sinking sort. From our tests, Sinking sort took up the most execution time and shattered the other algorithms in terms of average inlcusive time, revealing to us that as N goes up, Sinking sort’s performance must get worse [5][6][7][8]. The random and ordered data fail to yield a pattern of most successful algorithm, but the data in reverse order and the 10% random data give Radix sort a chance to shine, throughly beating out the other algorithms both in terms of inclusive percentage and time [7][8]. Ironically enough, Sinking sort appears to have the most difficulty at this N value of 1,000 when the data is already ordered in some way, with both the ordered and reversed data sets allowing Sinking sort to run for over 60% of the program’s execution [5][6]. Lastly, N = 10,000 once again awards Sinking sort with the title of most inefficient. Sinking sort took up a minimum of 41.83% of execution time across all four scenarios, peaking at 84.65% in the random data set [9][10][11][12]. However, increasing N to a bulky 10,000 seems to have caused issues in several algorithms, with 6 algorithms not finding any execution time at all for the random data, and 3 not finding any time in the sorted data set [9][10]. This anomaly is accompanied by the fact that of the three N values tested, only N = 10,000 establishes a clear trend of the worst and best performing algorithms. Sinking sort is of course the worst, but Counting sort surprisingly emerges as the most efficient, with almost no presence on the percentage of the execution’s runtime and the smallest average inclusive time [9][10][11][12]. As an aside, Counting sort could be either extremely fast in these scenarios, or it is not fully completing or sorting the data in the way we would like, which would explain the low average time for each call as well as the minimal appearance on the percentage of runtime.

# Summary

In conclusion, we can see from this short report that even the slightest of changes or factors in a data environment can completely affect what algorithm might be the best to use. For larger data sets, we can draw from our own results that Sinking sort is typically the worst/slowest algorithm to use, while more “involved” algorithms like Radix and Counting can outperform the others. While it is easy to believe that less or more simplified steps for an algorithm would lead to a faster runtime, they can often create more work for the program. Performing a simple step thousands or hundreds of thousands of times will always be less efficient and overall take more time than performing a more complex operation a hundred times. For small data sets, the algorithm used does not necessarily matter too much. In most use cases, the differences in performance are microscopic and would only be noticed through observation in a profiling environment, like we have done here. For larger sets, algorithms like Counting and Radix might give the best performance, but other options like Merge sort and Quicksort, both the original and Median of Three methods, can also excel as an efficient algorithm to get the job done. While not the worst performing, Selection sort stood out as a close next-to-last option, many times trailing only second to Sinking sort for the highest inclusive percentage but still performing well in other scenarios.

# Appendix 1 – Data and Graphs

Graphical user interface, application, table, Excel

Description automatically generated

[1] Table for N = 100 and List filled with random integers, unsorted

Graphical user interface

Description automatically generated with low confidence

[2] Table for N = 100 and List filled with data in order

Graphical user interface

Description automatically generated with medium confidence

[3] Table for N = 100 and List filled with data in reverse order

Graphical user interface, application, table

Description automatically generated

[4] Table for N = 100 and List filled with 90% ordered data, 10% random

Graphical user interface, application, Excel

Description automatically generated

[5] Table for N = 1,000 and List filled with random data

Graphical user interface

Description automatically generated with low confidence

[6] Table for N = 1,000 and List filled with data that is already in order

Graphical user interface, application

Description automatically generated

[7] Table for N = 1,000 and List filled with data in reverse order

Chart

Description automatically generated with low confidence

[8] Table for N = 1,000 and List filled with data that is 90% ordered, 10% random

Graphical user interface, chart, application

Description automatically generated with medium confidence

[9] Table for N = 10,000 and List filled with random data

Application

Description automatically generated with medium confidence

[10] Table for N = 10,000 and List filled with data that is already in order

Graphical user interface, chart, application

Description automatically generated

[11] Table for N = 10,000 and List filled with data that is in reverse order

Graphical user interface, application

Description automatically generated

[12] Table for N = 10,000 and List filled with data that is 90% ordered, 10% random

# References

[1] Refers to Graphic 1

[2] Refers to Graphic 2

[3] Refers to Graphic 3

[4] Refers to Graphic 4

[5] Refers to Graphic 5

[6] Refers to Graphic 6

[7] Refers to Graphic 7

[8] Refers to Graphic 8

[9] Refers to Graphic 9

[10] Refers to Graphic 10

[11] Refers to Graphic 11

[12] Refers to Graphic 12