Profiling Research Paper

Analysis of Different Sorting Algorithms Using a Code Profiler

Course: CSCI 2210-002 - Data Structures

Author: Dakota Cowell

Table of Contents

[Introduction 3](#_Toc528793469)

[The Problem 3](#_Toc528793470)

[Solution Plan & Tasks Performed 3](#_Toc528793471)

[Results and Conclusions 4](#_Toc528793472)

[Summary 5](#_Toc528793473)

[Appendix 1 – Data and Tables 6](#_Toc528793474)

# Introduction

This research paper will give a brief introduction to the advantages and disadvantages of using different sorting algorithms. This will cover the most common sorting algorithms and the common uses for each of them. Each sorting algorithm was tested using random data, sorted data, reverse sorted data, and nearly sorted data. Each algorithm was also tested on lists of different sizes. These variables will show how different algorithms should be used in different situations.

# The Problem

There are many different algorithms that can be used when sorting data. Because there are so many different algorithms, it becomes difficult to determine which algorithm is the best one to use. Developers must examine many different factors to determine which algorithm to use. If a developer uses the incorrect sorting algorithm, it may cause the program to be considerably slower. This requires developers to be more aware of which algorithms they’re using when designing software.

# Solution Plan & Tasks Performed

Several tests were performed to determine which algorithms are most appropriate for different situations. To test the performance of each algorithm, a solution was written in C# using Visual Studio to simulate data being sorted. To accurately compare the performance of the algorithms, the same data was used for every sorting algorithm.

The algorithms used in this performance test are the most common algorithms used in the software development industry. The algorithms that were tested are the sinking sort, also known as the bubble sort, the selection sort, the insertion sort, the merge sort, the quick sort, the median-of-three quick sort, the shell sort, the counting sort, and the radix sort. The effectiveness of each of these sorts depends on the situation that the sort is needed.

To judge the usefulness of each algorithm, several variables were changed between runs. The algorithms were performed on different amounts of data and on data that is in different sorted order. Each algorithm was performed on data sets of one hundred integers as well as data sets of one thousand integers. Originally, the plan was to perform the algorithms on data sets of ten thousand as well, but some of the algorithms are recursive, which caused the program to run out of memory. This was done to show how different sorts are more effective with fewer amounts of data, and some sorts are more effective or more data. Each algorithm was also tested on data sets that were in random order, reverse order, sorted order, and almost sorted order. These variables show how some algorithms are better than others when the information is in a different order.

After each run of the program, Visual Studio’s code profiler was used to analyze how well each algorithm performed on each data set. The data was then recorded into Microsoft Excel for easy comparison of the different algorithms. The raw data can be found in Appendix 1.

# Results and Conclusions

When analyzing the results of the algorithms’ performances, it is usually easiest to examine the percentage of time that each algorithm took in comparison to the other algorithms. The tables in Appendix 1 show the percentage of time that each sort took as well as the average elapsed inclusive time. Several conclusions can be drawn from examining the data.

The sinking sort always had the worst performance on the reverse sorted list, while the shell sort and the counting sort had the best performance on the reverse sorted lists[[1]](#footnote-1). Although Big O of the counting sort and the sinking sort are both O(n2), shell sort had the best performance with sinking having the worst. This may be happening because the sinking sort has to swap every single value in the array, while the shell sort only has to swap a few. This shows that Big O notation does not always show what sorting method would be the most efficient. As a developer, you should never use a sinking sort when you know all of the data is in reverse order.

The selection sort performed poorly on every set of data. It was always in the top 4 of least efficient algorithms. Big O of the selection sort is O(n2). Although this is better than larger exponents, it is best to use an algorithm that is more efficient than O(n2). Many of the algorithms tested can be as efficient as n log(n). Selection sort seems to be the least useful sort out of the ones tested, but it may be easier to program than some of the other sorts.

For a list of one hundred integers, the quick sort performed better than the median-of-three quick sort on every ordering of data. Although the median-of-three quick sort is supposed to be an improvement of the quick sort, sorting one hundred values was always slower when using the median-of-three method[[2]](#footnote-2). The advantage of the median-of-three method can be seen when you observe tables 5 through 8. When using the median-of-three method to sort 1000 integers, the quick sort method is considerably slower. The quick sort was one of the least efficient sorting algorithms when attempting to sort a sorted list, but the median-of-three method was one of the most efficient. This shows that the quick sort should be used on small sets of data, and the median-of-three method should be used on larger sets.

The sinking sort was very efficient for sorted values, but it was the least efficient algorithm when sorting larger data sets. When examining the data, you will see that the sinking sort took up about 59% of the total time when sorting random data[[3]](#footnote-3). It also took up about 55% of the time when sorting reverse ordered data and about 37% of the time when sorting almost sorted data[[4]](#footnote-4). This seems to be because the sinking sort performs many swap operations when there are large quantities of data. Many of the other algorithms have the same Big O complexity, but they may not all have to swap the information as many times as the sinking sort. Most of the other algorithms perform many comparisons and few swaps. Overall, the sinking sort is one of the least efficient algorithms unless you are sorting data that is already sorted.

The counting and shell sorts performed very well on almost all of the tests whether the quantity of data was large or small. Although the counting sort performed well in the tests, it may not actually be as efficient as it seems. The random numbers being generated are only between 0 and 99 because when they were higher, the program ran out of memory. If the numbers in the list were larger, the counting sort may be less efficient because the Big O of it is O(n + k). Developers should use counting sort when the values being stored are small. The shell sort was also one of the most efficient sorts being tested. The only test it performed poorly on was sorting one thousand almost sorted integers[[5]](#footnote-5). With the shell sort, the data is broken up into bins and then sorted using a gap. It is useful because unlike the bubble sort, it does not have to swap every value that is next to each other. It swaps data between the gap. The shell sort can be used in many different cases, but it is not the most effective sort to sort almost sorted data.

# Summary

There are many different sorting algorithms, and almost all of them have situations where they could be the most efficient algorithm to use. The performance of each algorithm can be shown theoretically using Big O and Big Omega. The data shows that the most efficient Big O algorithm doesn’t always imply that an algorithm is the most efficient. Other factors need to be considered as well as Big O. Some of the other things that can be considered are the average complexity as well as the best-case scenarios. Some of the algorithms will always perform at best-case if you already know the order.

There’s not one “best” algorithm for sorting data, but there can be a best algorithm for certain situations. Algorithms should be chosen depending on how large the data set is and how sorted the data is. Some algorithms such as the selection sort and the sinking sort never perform better than everything else and should only be used in very specific situations. Having an understanding of the different algorithms and how they work on different data sets allows developers to write much more efficient code and create better programs.

# Appendix 1 – Data and Tables

|  |  |  |
| --- | --- | --- |
| **100 Randomly Sorted Ints** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Merge | 6.86% | 1.47 |
| Sinking | 5.90% | 1.27 |
| Radix | 4.83% | 1.04 |
| Selection | 3.67% | 0.79 |
| Median-Of-Three Quick | 2.19% | 0.47 |
| Quick | 0.91% | 0.19 |
| Insertion | 0.73% | 0.16 |
| Shell | 0.31% | 0.07 |
| Counting | 0.38% | 0.08 |

Table 1 – Sorting 100 Randomly Ordered Integers

|  |  |  |
| --- | --- | --- |
| **100 Sorted Int Values** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Radix | 5.35% | 1.05 |
| Selection | 3.95% | 0.78 |
| Median-Of-Three Quick | 3.76% | 0.74 |
| Merge | 2.96% | 0.58 |
| Quick | 1.64% | 0.32 |
| Counting | 0.44% | 0.09 |
| Shell | 0.23% | 0.05 |
| Insertion | 0.05% | 0.01 |
| Sinking | 0.01% | 0.00 |

Table 2 – Sorting 100 Sorted Integer Values

|  |  |  |
| --- | --- | --- |
| **100 Reverse Sorted Int Values** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Sinking | 9.06% | 2.05 |
| Merge | 5.86% | 1.32 |
| Radix | 4.48% | 1.01 |
| Selection | 4.48% | 1.01 |
| Median-Of-Three Quick | 1.88% | 0.42 |
| Insertion | 1.71% | 0.39 |
| Quick | 1.47% | 0.33 |
| Counting | 0.35% | 0.08 |
| Shell | 0.22% | 0.05 |

Table 3 – Sorting 100 Reverse Ordered Integer Values

|  |  |  |
| --- | --- | --- |
| **100 Almost Sorted Int Values** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Merge | 6.63% | 1.47 |
| Radix | 4.87% | 1.08 |
| Selection | 3.17% | 0.70 |
| Sinking | 2.33% | 0.52 |
| Median-Of-Three Quick | 2.08% | 0.46 |
| Quick | 1.95% | 0.43 |
| Insertion | 0.51% | 0.11 |
| Counting | 0.41% | 0.09 |
| Shell | 0.29% | 0.06 |

Table 4 – Sorting 100 Almost Sorted Integer Values

|  |  |  |
| --- | --- | --- |
| **1000 Randomly Sorted Ints** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Sinking | 59.10% | 49597.25 |
| Selection | 15.39% | 7631.09 |
| Insertion | 15.32% | 7596.76 |
| Merge | 10.15% | 5032.22 |
| Radix | 0.00% | 2.15 |
| Quick | 0.00% | 1.58 |
| Median-Of-Three Quick | 0.00% | 1.24 |
| Shell | 0.00% | 0.91 |
| Counting | 0.00% | 0.12 |

Table 5 – Sorting 1000 Randomly Ordered Integer Values

|  |  |  |
| --- | --- | --- |
| **1000 Sorted Int Values** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Quick | 62.56% | 15423.44 |
| Selection | 27.24% | 6715.38 |
| Merge | 0.01% | 2.05 |
| Radix | 0.01% | 1.66 |
| Median-Of-Three Quick | 0.00% | 1.16 |
| Shell | 0.00% | 0.43 |
| Counting | 0.00% | 0.20 |
| Sinking | 0.00% | 0.07 |
| Insertion | 0.00% | 0.07 |

Table 6 – Sorting 1000 Sorted Integer Values

|  |  |  |
| --- | --- | --- |
| **1000 Reverse Sorted Int Values** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Sinking | 55.05% | 46832.60 |
| Insertion | 17.96% | 15281.49 |
| Selection | 9.02% | 7677.70 |
| Quick | 8.98% | 7642.69 |
| Merge | 8.96% | 7619.13 |
| Radix | 0.00% | 2.58 |
| Median-Of-Three Quick | 0.00% | 0.85 |
| Counting | 0.00% | 0.38 |
| Shell | 0.00% | 0.28 |

Table 7 – Sorting 1000 Reverse Sorted Integer Values

|  |  |  |
| --- | --- | --- |
| **1000 Almost Sorted Int Values** | **Inclusive % Time Elapsed** | **Avg Elapsed Inclusive Time** |
| Selection | 42.81% | 15365.12 |
| Sinking | 37.50% | 13458.65 |
| Shell | 19.56% | 7019.95 |
| Merge | 0.00% | 4.24 |
| Quick | 0.01% | 2.29 |
| Radix | 0.01% | 2.02 |
| Insertion | 0.00% | 1.53 |
| Median-Of-Three Quick | 0.00% | 0.85 |
| Counting | 0.00% | 0.22 |

Table 8 – Sorting 1000 Almost Sorted Integer Values

References

Bailes, Don. “Lecture 12 – Sorting Algorithms.” East Tennessee State University, 31 October 2018, Lecture

1. This can be observed in tables 3 and 7 in Appendix 1 [↑](#footnote-ref-1)
2. This can be observed in tables 1-4 in Appendix 1 [↑](#footnote-ref-2)
3. This can be observed in Table 5 in Appendix 1 [↑](#footnote-ref-3)
4. This can be observed in Tables 7 & 8 in Appendix 1 [↑](#footnote-ref-4)
5. This is shown in Table 8 [↑](#footnote-ref-5)