Profiling Research Paper

Analysis of Sorting Algorithms Using a Code Execution Profiler

Course: CSCI 2210-201, Fall 2016

Author: Allison Ivey

Contents

[Introduction 2](#_Toc465287391)

[Algorithms Used 2](#_Toc465287392)

[The Problem 4](#_Toc465287393)

[Solution Plan 4](#_Toc465287394)

[Tasks Completed 5](#_Toc465287395)

[Results and Conclusions 5](#_Toc465287396)

[Summary 7](#_Toc465287397)

[Appendix 1 – Data and Graphs 8](#_Toc465287398)

[References 19](#_Toc465287399)

# Introduction

Not all sorting algorithms are created equal. Computer scientists need to be able to estimate the amount of work an algorithm must do and how long it will take to run. The range expect for run time falls between an upper and lower bound. These bounds are influenced by how much data needs to be processed. Very few data items can be sorted quickly and easily. The challenge comes when an algorithm must be selected to sort large amounts of data.

Algorithms can be evaluated using three metrics; run-time performance, memory usage, and ease of understanding. Run-time performance is defined by how long an algorithm runs in order to do a particular job. Memory usage refers to the resources that are required to hold the amount of data that needs to be processed. Ease of understanding, programming, testing, and maintenance need to be considered when selecting an algorithm for a problem. The more complex the code the harder it is to maintain. It is important that the code is easy to understand and update. Many times, complex algorithms that run very quickly use more resources to do so. Algorithms that are easy to program and test run slower because of a lack of complexity. These trade-offs need to be considered in order to select the best algorithm for the job at hand.

Algorithms can be tested both empirically and theoretically. Empirical evaluations use test suites to enter data into the system, gather information about the algorithm, and make decisions as to their effectiveness. There are many tools available to gather this type of information. For this project, we used Visual Studio’s built in profiler. The program ran separately for each data set to allow for a comparison rather than a composite of all the data. In this way, we could collect empirical test data on each of the given algorithms.

Theoretical analysis does not require programs to be written rather, it requires an estimate of the amount of work algorithms must do to solve problems. A counting approach is used to estimate the average, best case, and worst case performance of an algorithm. Commonly the worst-case performance is the one that gets evaluated because it is easier to determine and illustrates the worst-case scenario. The worst-case scenario is based on the amount of data that needs to be processed and the number of steps to process that data. In this study the most interesting results came from testing larger amounts of data. All the algorithms performed relatively well given small amounts of data, but when larger amounts of data were introduced differences between the performance of the algorithms became more apparent as illustrated by their order of magnitude.

To understand the worst-case scenario for an algorithm the big O complexity of the algorithm is evaluated. Figure 1 illustrates that the greater the slope of the big O value the greater the number of operations needed to complete the algorithm. Ideally, the order of magnitude remains low so that no matter how much data is entered in to the system the number of operations to sort the values does not grow at a rapid rate.

# Algorithms Used

There are many different types of algorithms. Some are better than others depending on the context in which they are used. No algorithm is going to work the best in all cases. For the purposes of this study we looked at seven different sorting algorithms (Mishra, 2008).

Sinking Sort

Sinking sort takes the first number in a set of numbers and compares it to the one directly after it. If the first number is greater than the second number, the numbers are swapped. If the numbers are already in order the first number is left in place and the second number is compared to the one right after it. This continues until the highest number “sinks” to the bottom of the list hence this algorithm being called sinking sort. The first pass requires the algorithm to make n-1 comparisons where n is the number of elements in the list. On the second pass the algorithm is required to make n-2 comparisons. This goes on until the user makes .5N^2+.5N comparisons. In this equation N^2 dominates so sinking sort is said to have a big O of N^2. This can also be seen through the code for the sinking sort. This algorithm uses nested loops meaning that n operations must be done n times or n^2 (KARIMIZADEH, 2015).

Insertion Sort

In insertion sort the first number in the list is compared to the second number in the list. If it is larger the lower number is inserted above the higher number. The next number is then compared to those two values and put in the proper place relative to the preexisting two numbers. This continues until all the numbers are in their correct order. This algorithm uses nested loops so it has a big O of of N^2 (KARIMIZADEH, 2015).

Selection Sort

Selection sort goes through the entire list and finds the largest number. When it finds the largest number, it swaps it with the number with the highest index. The list is then gone over again and the second highest number if found. When the number is found, the number is swapped with the number at index n-1. This continues until all the numbers have been compared and are in numerical order. This algorithm also uses nested loops so it is thought to have a Big O of N^2 (KARIMIZADEH, 2015).

Quick Sort and Quick Sort Median of Three Pivot Selection

Quick sort is also called pivot sort. A number is selected from the list. That number is compared with the first number in the list. If the number is higher, it is swapped with the pivot and the algorithm then looks at the next value in the list of numbers and compares it to the last number in the list. If the last number is greater than the second number in the list they are swapped. If they are not, then the second number stays in its place and the third number then becomes the pivot. This continues until the original pivot is placed in its correct spot. The groups are then separated and the process begins again with two separate lists from the same list. If there are less than 10 values when the items are split many times the programmer will sort the small amount of values using an easier and less memory intensive algorithm to sort the two hemispheres. Quick sort median of three pivot selection is included in the same section as quick sort because it uses the same strategy for sorting the elements, but it finds a better pivot to start the program with. It does this through selecting the first number in a list, middle number in the list, and the last number in the list. These three numbers are compared and the middle number is selected. This insure that the highest or lowest value in the list is not selected and there is a higher likelihood that the number that is selected will be in a more optimal starting position. These two sorts have an average big O of n log n (KARIMIZADEH, 2015).

Shell Sort

Shell Sort is like quick sort but with some distinct differences. Shell sort begins by selecting a gap size. Once a gap size is established the first term and the term at the end of the gap are compared. If the value of the first number is greater than the number selected with the gap number, then the values are swapped. If they are not, they stay in their place and the algorithm moves on to comparing the second value in the list with the number at the end of the gap. This goes on until the entire list has been evaluated. Once all the values using the gap have been swapped a smaller gap is established and the same process repeats itself until the entire list is in order. Shell sort also uses nested loop so its big O value is N^2 (KARIMIZADEH, 2015).

Merge Sort

This approach also uses the strategy of dividing the list of numbers into parts, but this approach has proven to be much faster. The values in the list are split until they are singletons. Once they are singletons they are grouped in twos and compared with one another. If they are out of order they switch positions. If they are not, they stay where they are. This is done with all the pairs of numbers in the list. Once this is done the pares are merged with the pare of numbers right next to it. The numbers in that set are then compared and numbers are moved around based on their value. The algorithm continues to merge back together while sorting the values in the list. This type of sorting has proven to be one of the faster algorithms for sorting numbers. This algorithm has a big O of N log N (KARIMIZADEH, 2015).

# The Problem

How do the run-times of each of the algorithms compare given 10, 100, or 1000 in order, semi out of order, or random values.

# Solution Plan

* A program was written that contained all seven sorting algorithms; insertion sort, merge sort, shell sort, quick sort, quick median of three sort, selection sort and sink sort.
* Each of the sorting algorithms were passed the same list of values as an argument.
  + During the first pass the list was populated with 10 values of ordered data.
  + During the second pass the list was populated with 10 values of semi ordered data.
  + During the third pass the list was populated with 10 values of randomly ordered data.
  + During the fourth pass the list was populated with 100 values of ordered data.
  + During the fifth pass the list was populated with 100 values of semi ordered data.
  + During the sixth pass the list was populated with 100 values of randomly ordered data.
  + During the seventh pass the list was populated with 1000 values of ordered data.
  + During the eighth pass the list was populated with 1000 values of semi ordered data.
  + During the ninth pass the list was populated with 1000 values of randomly ordered data.
* After each change in the list type Visual Studio’s Profiler was used to assess the efficiency of each of the seven algorithms.
* The information was then recorded using Microsoft Excel.

# Tasks Completed

* Program was written
* Sorting methods were fed lists with different sizes and orders
* Profiler was run after each change in list type and size.
* Information was transferred to excel for synthesis.
* Graphs were built to understand the relationship between list size, list order, and run time.

# Results and Conclusions

Merge Sort

Although merge sort consistently had one of the highest runtimes, the amount of time that the sort took did not vary significantly given different size lists of numbers. This means that no matter what size data the algorithm assessed or what order the data was in, it had a predictable runtime. This algorithm would work well with large data sets that are in varying degrees of order. Using this algorithm would suggest that low runtimes could be expected with large data sets relative to the other algorithms. This algorithm may not be the best choice out of all the other algorithms for small data sets due to its high run-time values in this context. (Figure 2)

Insertion Sort

When insertion sort was given a small list, it performed about the same no matter what order the data was in. Once the data set started to get larger the algorithm started to show some differences when it came to the order of the values. When the values were random it took the algorithm far longer to run than it did when the values were in semi order or order. This supports the fact that it has a big O of N^2. As the job got harder the algorithm took longer to sort. (Figure 3)

Original Quick Sort

Quick sort much like merge sort showed little change in the runtime when given data in different orders and lists of different sizes. It handled the data consistently and predictably. The amount of data did not seem to slow or speed up the algorithm. (Figure 4)

Quick Sort Median of Three Sort

These results did not behave as expected. It was expected the run times would stay consistent no matter what order the data was in and no matter what size the list was. It was expected to perform much like the quick sort. The results do not support this conclusion. It seems that there were some consistencies in run time between the list that had ten values and the list that had 100 values but there was a big jump in run time for the list with 1000 random values. It took longer than any of the other algorithms. (figure 5)

Selection Sort

Selection sort consistently took longer to sort data that was randomized but could sort data that was in order or semi in order relatively quickly. The run times at a list size of 10 or 100 values was relatively comparable to the other algorithms. When the value of N spiked so did the run time of the selection sort putting it in the top three slowest algorithms for unsorted data. (figure 6)

Shell Sort

Shell sort performed very well compared to the other algorithms in most areas. When it came to the semi-ordered data there was a surprising result. It seems that the random data that it was assigned gave it a very high value for the run time of 1000 values. This data did not seem to match the other run time values that the model produced. Other than this strange outcome shell sort performed comparable or even fast as compared to the other algorithms. (figure 7)

Sink Sort

Sink sort consistently had higher run times than the other algorithms in the program. As the amount of values in the list got bigger the run times got longer. The order of the values did not seem to make a difference when the amount of values in the list were small. When the list size got bigger the discrepancy between the runtimes of ordered data verses unordered data began to show. Sink sort performed the worst out of all the algorithms once the data set got larger than 100. (figure 8)

Comparing All the Sorting Methods

The run times for all the methods were relatively comparable to one another with ordered data inside both big and small lists except for selection sort. Selection Sort took far longer than any other algorithm when given 1000 ordered values. When it came to data that was semi in order again there were not that many differences between the different algorithms apart from insertion sort, selection sort, and shell sort having very high values when the size of the list rose to 1000. Differences also begin to appear when we looked at the data for the lists with 1000 values. When random data was sorted using Merge sort and the original quick sort very good results occurred no matter the size of the list. These algorithms seem to perform well for large sets of data in any given order. (figure 9)

Number of Calls

The number of calls graph is very telling. When the list size got larger the amount of calls for the randomly ordered values went up as compared to ordered values and semi ordered values. The more in order the data was the lass calls it had to make. When the list size was small the differences in the number of calls was not that noticeable. As the number of values in the list begin to go up discrepancy between ordered and unordered values began to appear. (figure 10)

# Summary

Run-times were comparable when the size of the lists were small. When the size of the lists increased, the algorithms performed differently than one another. The algorithms that split the data up, sorted the parts separately, and then recomposed the parts once sorted performed much better than those that went through the list swapping numbers one at a time. Although it may seem logical to choose the algorithm that performed the fastest, speed is not the only metric that an algorithm must be measured against. When an algorithm uses a method that splits the data up into parts, the data has to be sorted and stored separately then recomposed. This takes a lot more memory to manage than algorithms that simply switch values. Large lists require more memory, if the computer is not able to accommodate this amount of memory the program will not be able to function or it will slow way down. Like anything, algorithms have their relative strengths and weaknesses. Some of them are simple and slow, some of them are fast and complex, others are somewhere in between. What algorithm to choose can only be decided once the problem the algorithm must solve has been established.

# Appendix 1 – Data and Graphs

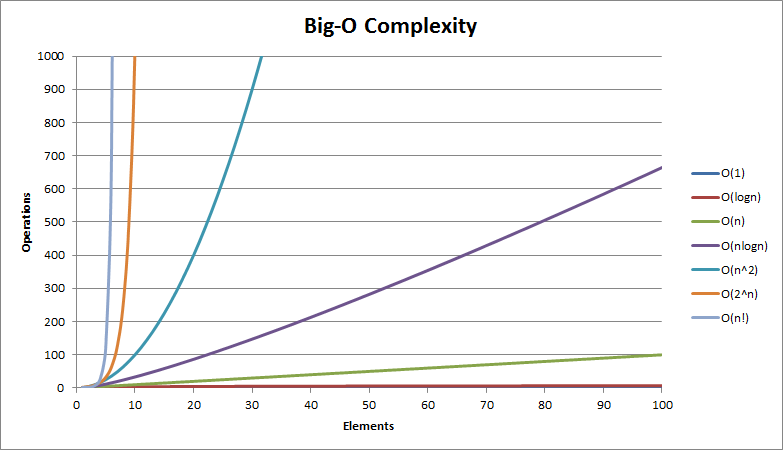


Figure 1 Big O Notation



Figure 2



Figure 3



Figure 4



Figure 5



Figure 6



Figure 7



Figure 8

Figure 9



Figure 10

# References

KARIMIZADEH, M. M. (2015). Enhancing and Optimization Sorting Algorithms: An Empirical Study. *Cumhuriyet Science Journal 36.6* , 354-266.

Mishra, A. D. (2008). Selection of best sorting algorithm. *nternational Journal of Intelligent Information Processing 2.2* , 363-368.