

Relationship between Time Complexity and Number of Elements, within different Sorting Algorithms.

Mino Karadzhov and Kestutis Dikinis

December 10th 2021

Module: SEAR

Venlo, Limburg, Netherlands

Abstract

Sorting collections of information is one of common usages of computers nowadays. Sorting operations could take place in many of our daily interactions with information systems. There are several different techniques for conducting a sorting operation and choosing the proper one can reap huge savings and even make some problems possible to be solved in general[1].

One of the metrics for measuring Algorithm Performance is the Time Complexity, which corresponds to the Time required for a given algorithm to fully complete it's task. In our case, this means fully sorting a collection. There is a number of factors that can affect the Time Complexity in both negative and positive way. The number of elements for sorting is one such factor, but what is the exact way that it affect the Time Complexity of a Sorting Algorithm?

This Paper describes an experiment, under the form of an Empirical analysis of four different Sorting Algorithms(SelectionSort, HeapSort, BubbleSort and Quicksort), that has a goal of studying the relationship between Time Complexity and the Number of elements for sorting. The results show an overall positive relationship between the two variables. However, the experimental results also show us that the exact ratio of growth is not the same for every Sorting technique.

Contents

1	Introduction	1
1.1	Context and Background	1
1.2	Modern Usage, developments	1
1.3	Sorting algorithm explanations	1
1.4	Hypothesis, Problems and Considerations	2
2	Methodology	3
2.1	Working Implementation of our Algorithms.	3
2.2	Developing and using our experimental environment.	3
2.3	Data Analysis and Visualisation	4
3	Results	5
3.1	Experimental Results	5
4	Discussion	7
4.1	Interpretation	7
4.2	Considerations	7
4.3	Evaluations	8

List of Figures

1	Figure 3.1	5
2	Figure 3.2	5
3	Figure 3.3	6
4	Figure 3.4	7
5	Figure 3.5	8

1 Introduction

The Purpose of this section is to introduce what is sorting algorithms and history of them, so there is general understanding about what is sorting algorithm. Introduce sorting algorithms used in research to get principle of working of different algorithms. And most importantly introduce the main research question. Also, this section will discuss problems, considerations and the hypothesis. The main question of this research is: How does the time complexity of different sorting algorithms changes with increasing number of elements? It is important to learn about time complexity of sorting algorithms as this discipline gives us the potential to reap huge savings, even to the point of enabling us to do tasks that would otherwise be impossible. [1]

1.1 Context and Background

In computer science, a sorting algorithm is an algorithm that takes elements of a list and puts them into an order - ascending or descending. The most popular values used are numerical and lexicographical order. It is important to have efficient sorting for optimising other algorithms, such as merge and search algorithms, as they require data to be in sorted list. Whenever there is a problem to be solved, there are many ways to approach it. Even though all those approaches end up with the same result, the time it takes for each approach can be vastly different. And the best example of that is sorting algorithms.

1.2 Modern Usage, developments

Sorting algorithms are everywhere in day-to-day life in digital world. Everywhere starting from file browser in a computer - you can sort files on any parameters. Galleries in mobile phones - images sorted by date, contacts sorted alphabetically . Searching for a product in online store will have an option to sort products in many ways. Commercial computing: government organisations, financial institutions, and commercial enterprises organise much of this information by sorting it. Whether the information is accounts to be sorted by name or number, transactions to be sorted by time or place, mail to be sorted by postal code or address, files to be sorted by name or date, or whatever, processing such data is sure to involve a sorting algorithm somewhere along the way. [2] Numerical computations: Scientific computing is often concerned with accuracy (how close are we to the true answer?). Accuracy is extremely important when we are performing millions of computations with estimated values such as the floating-point representation of real numbers that we commonly use on computers. Some numerical algorithms use priority queues and sorting to control accuracy in calculations. All in all, all any and every kind of list there is has been sorted in some way or another, as it is hard for a human to make use of unsorted data.[2]

1.3 Sorting algorithm explanations

Bubblesort is a simple sorting algorithm that belongs to the family of comparison sorting. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. Bubblesort has a worst-case complexity

$O(n^2)$ and in the best case $O(n)$. Its memory complexity is $O(1)$. [2]

Heapsort is a comparison-based sorting algorithm, and is part of the Selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quicksort, it has the advantage of a worst-case $O(n \log n)$ runtime.[2]

Mergesort belongs to the family of comparison-based sorting. It has an average and worst-case performance of $O(n \log n)$. Unfortunately, Mergesort requires three times the memory of in-place algorithms such as Insertionsort. [2]

Selection sort belongs to the family of in-place comparison sorting. It typically searches for the minimum value, exchanges it with the value in the first position and repeats the first two steps for the remaining list. On average Selection sort has a $O(n^2)$ complexity that makes it inefficient on large lists. Selection sort typically outperforms Bubblesort but is generally outperformed by Insertionsort.[2]

On the other side of the sorting algorithm spectrum there is Bogosort. It was designed as a joke about how not to design sorting algorithms. Bogosort takes and randomly swaps places all elements of the list and repeats it until the list is sorted. Complexity of Bogosort algorithm is $O((n+1)!)$.

1.4 Hypothesis, Problems and Considerations

Derived from main research question, we can make a hypothesis that we are going to see a growth of the Time Complexity, towards Size of Collection.

1.4.1 Time Complexity and the importance of it.

Time complexity or more known as big O notation is a equation that describes how the runtime scales in respect to input variables. As time complexity describes the relationship between runtime and input variables it can be said, that big O notation is used to have a rough approximation of how the program will react to being scaled and when it is used to solve larger problems. [7]

1.4.2 Factors, operating on Time Complexity.

Determining the time complexity of an algorithm is important, and is done by few steps:
-Breaking algorithm into individual operations. -Calculating big O for each calculation.
-Adding up big O of each operation. -Removing the constants. -Finding the highest order term. After following these steps algorithms big O is found.

There are few common cases of big O: $O(1)$ - determining if number is odd or even. $O(\log n)$
- Finding an item in a sorted array using a binary search. $O(n^2)$ - Bubble sort or Insertion sort. $O(n!)$ - Solving the traveling salesman problem using just brute-force search.

1.4.3 Different methods of approaching our Research question.

2 Methodology

In order to either confirm or reject our initial hypothesis, an empirical analysis of certain sorting algorithms was conducted. This chapter of the paper describes the process of conducting the experiment and creating the proper experimental environment. The subject group of sorting algorithms includes: QuickSort, BubbleSort, SelectionSort and HeapSort. The main goal of the analysis is to observe changes of the time complexity and memory usage, when the same algorithm is given the task to sort a collection of a certain size and type. Each observation of our experiment includes a task of sorting, performed by one of the sorting algorithms from our subject group, performed on a collection of certain size, containing either numerical or character type of elements.

In order to conduct the Empirical Analysis, a custom-made Java application was developed and used in the process. The main purpose of which is to create a proper experimental environment, that is going to allow close observation of each algorithm from our subject group.

The overall goal is to gain an output, which allows data analysis on the behaviour of time complexity and memory used when the size and type, of the collections for sorting, change.

2.1 Working Implementation of our Algorithms.

One requirement of the empirical analysis is a working implementation of the Sorting Algorithms, which are going to be used for the experiment. Due to this, the Java programming language was used to create working implementations of each one of the four members of our subject group. All of the concrete implementations of the algorithms implement a common interface, towards which the testing is being conducted.

2.2 Developing and using our experimental environment.

In order to serve the need of a proper experimental environment for conducting the Analysis, Petko was developed. Petko is the name of the custom-made algorithm analysis tool, that generates unsorted collections of a fixed size and uses sorters(Sorting Algorithms) to conduct the sorting operation. The latter is being closely monitored and both the time and memory, required for each sorting is being recorded in an output file..

Each Algorithm is used to perform 36 different observations in two separate stages. Each Observation requires the algorithm to sort an unsorted collection of a pre-defined size and type. The size of the unordered collection grows with each following observation. During the first stage of observation, each algorithm is being used for performing an operation of sorting on numerical collection with number of elements 2, whereas the last(18th) observation provides the Algorithm with an unsorted collection of 262144 elements.

The second stage of the experiment, follows the same procedure as the first one. The main difference is the type of collections for sorting. During this stage, all of the unsorted collections are consisted of character type of elements. The growth of Collection Size is Logarithmic with base 2. Respectively, this means that the first collection for sorting is

with size 2^1 , whereas the collection for the last observation has a size equal to 2^{18} . The values within the unsorted collections were numeral

Petko actively uses the Java.time and Java.Runtime integrated libraries, in order to measure the Duration of time and memory used, between the start and end of the sorting process. Capturing the time complexity is made possible, by placing an Instant at the Start and End point of each Algorithm execution. The following PseudoCode provides a brief description of the technique:

Algorithm 1 Capturing time complexity within Petko

```

startPoint  $\leftarrow$  Instant.now()  $\triangleright$  Placing a start point, right before the start of the sorting
sort()
endPoint  $\leftarrow$  Instant.now()  $\triangleright$  Placing an end point, right after sorting
timeComplexity  $\leftarrow$  Duration.between(startPoint, endPoint)

```

In order to get a record of the used memory, Petko is using the Runtime java library. This allows us to interact with the application environment, where Petko is running. This is the same environment, where the sorting is taking place. The amount of used memory before the sorting is being recorded, as well as the amount of used memory after the sorting is being done. The following pseudocode gives a description on the technique.

Algorithm 2 Capturing memory complexity within Petko

```

usedMemoryBefore  $\leftarrow$  Runtime.totalMemory()  $-$  Runtime.freeMemory
sort()
usedMemoryAfter  $\leftarrow$  Runtime.totalMemory()  $-$  Runtime.freeMemory

```

Each Observation records the Sorting Algorithm that is being used, the time complexity of the observation, the Size of Collection, amount of memory used and the type of elements that are being sorted. This output is being gathered and recored into an external File under the .csv extension.

2.3 Data Analysis and Visualisation

Petko's output file provides a great base for conducting Data Analysis on the results of the observations. By doing this, it is possible to identify and study certain patterns in the covariation between time complexity, memory usage and the size of collection. In addition to this, it allows us to study the change of the same variables, when the sorting operation is being conducted on different type of elements.

The same output file was imported into R and the findings were visualised, thanks to the tidyverse library.

3 Results

This section provides a detailed overview of the experimental results.

3.1 Experimental Results

After conducting the emperical analysis, 144 datapoints were generated.

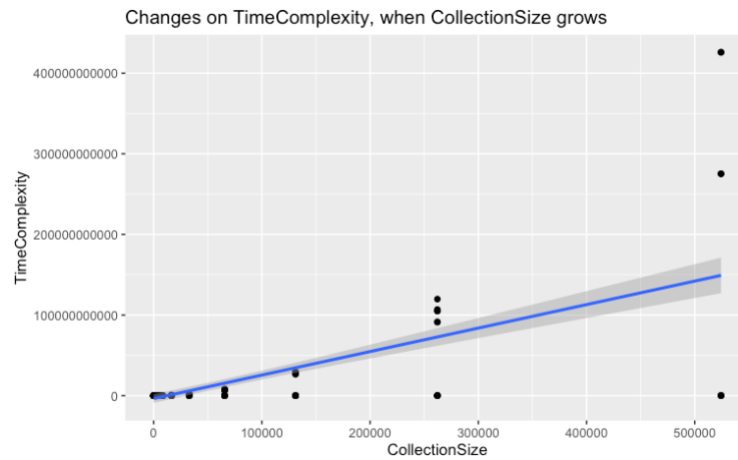


Figure 1: Overall view of correlation between TimeComplexity and SizeOfCollection

As we can clearly see from the above diagram, there is a positive relationship between TimeComplexity and the CollectionSize. In other words, the time required for sorting a given collection, increases as the size of the collection grows. Displayed here, is the general case, which includes both Numerical and Charecter type of collections.

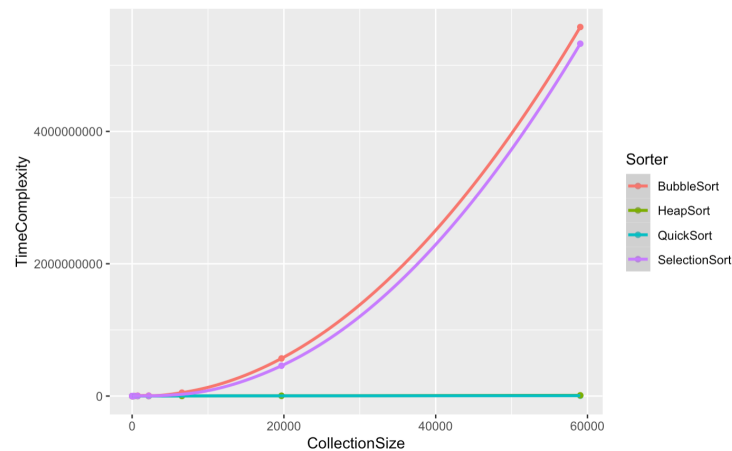


Figure 2: Overall view of correlation between TimeComplexity and SizeOfCollection, when sorting different type of elements

The above diagram provide us with the possibility to take a closer look at how the relationship between TimeComplexity and CollectionSize changes, when the sorting is applied on numerical and charecter type of elements. Looking at the graph, we can easily see that the growth is slightly bigger, when charecter elements are being sorted. Furthermore, this means that more time is required for sorting the same amount of charecter elements, compared to such of numerical type. We can also identify two outliers, which required significantly more time to be sorted. Both of these observations are being conducted on charecter type of collection.

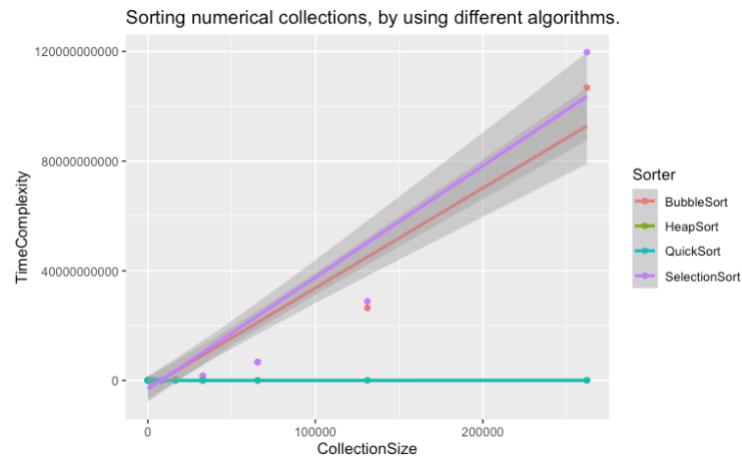


Figure 3: Overall view of correlation between TimeComplexity and SizeOfCollection, when using different sorting algorithms.

Taking a more focused look at the numerical collections, we can easily see that the growth of TimeComplexity is different for each algorithm. This means that the sorting operation takes a different amount of time, depending on the algorithm, used for the operation. In addition to this, we can see that the BubbleSort and SelectionSort take significantly more time, compared to HeapSort and QuickSort. The latter group is overlapping on the diagram, as their time complexity is in a big contrast, compared to the other one. From the above, we can identify two groups of algorithms - group of efficient algorithms and a group of less-efficient algorithms

From the above diagram, we can see that there is a similar positive relationship between RAMUsed and SizeOfCollection. This means that more memory is being used, when trying to sort bigger collections. We can see that the growth is sharper, compared to the one of the TimeComplexity.

The above figure shows us that there is a difference in the RAMUsed, when a different algorithm is being used for the sorting. This means that some algorithms require more memory to conduct the sorting than others. In the current case, we can clearly identify that QuickSort is more memory efficient, than the SelectionSort, which happens to be the least

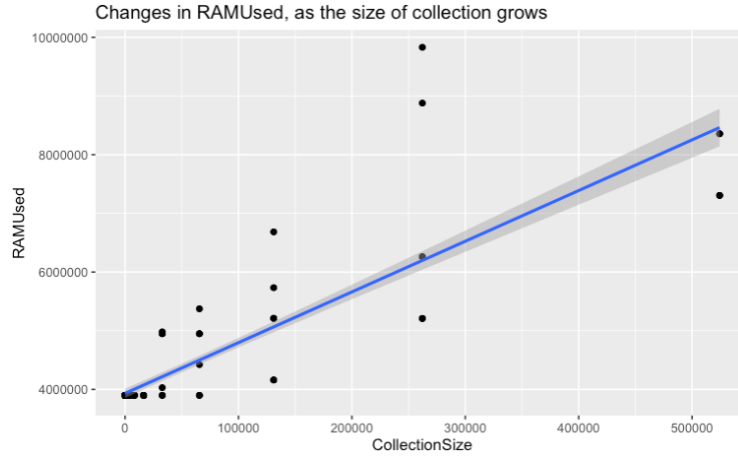


Figure 4: Overall view of correlation between RAMUsed and SizeOfCollection

efficient algorithm, when it comes to memory.

4 Discussion

In the following section we are going to discuss our results, give answer to our research question and provide a conclusion if our hypothesis was validated or not.

4.1 Interpretation

The Experimental results, clearly display a relationship, that can be described as a positive correlation between Time Complexity and Collection Size. This validates our initial hypothesis, as the increased number of elements for sorting clearly results in an increase of the time required for completing the sorting operation. In addition to this, we gained knowledge, that shows that different Sorting techniques (Algorithms) are not being affected in the same way by the increased number of elements for sorting. For instance, on the Overall view of our Experimental results, we can see that BubbleSort and SelectionSort are experiencing a drastic increase of their TimeComplexity, whereas the other two Sorters - QuickSort and HeapSort are not following the same pattern.

Based on this, we can conclude that the latter group of Sorters can be a better choice for executing sorting tasks, which are placing their focus on the Time Resource being spend.

4.2 Considerations

This subsection describes the Considerations made during the Experiment, as well as reasoning for some of the decisions described in the Methodology section.

4.2.1 Limitations of the Empirical type of Analysis

It has to be taken into account that the Empirical way of conducting the Analysis is not the best, in terms of time efficiency. That is due to the fact that this type of analysis, requires a

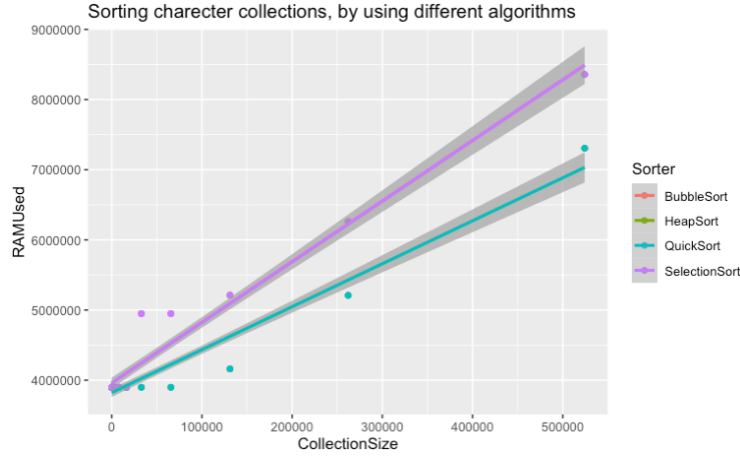


Figure 5: Overall view of correlation between RAMUsed and SizeOfCollection, when using different sorting Algorithms

working implementation of the subjects(Sorting Algorithms) that are going to be analysed. Furthermore, a certain amount of computing power is required, as this type of analysis requires the Algorithms to be put into action. In order to experiment with bigger Collections, which could be beneficial for revealing (otherwise hidden) patterns, larger amount of computing power is going to be required.

4.2.2 Limitations of the Collection Size

Because of the nature of Empirical Analysis, described in the last sub section, this experiment used Collections of size, just up to 2^{18} - 262144 elements. This limitation was present, mainly due to the fact that the machine, we used for testing, was not capable of experimenting with bigger collections. Would we have more computing power at our disposal, there could've been more patterns and datapoints to extend the scope of the experiment.

4.2.3 Limitations of the Machine, used for the Experiment

The described experiment was conducted on a Machine with the following technical specifications : CPU: 2 GHz Quad-Core Intel Core i5, 16 GB 3733 MHz LPDDR4X RAM. In addition to the Collection Size limitation, it is worth considering the fact that our experimental results were somehow influenced by the computational power, available with the above specifications. This means, that there is a high chance of observing different time complexity values, if the same experiment is conducted on another machine.

4.3 Evaluations

As this experiment is being conducted in a relatively small scope, taking into account the limited size of collections used, there could be an argument that this limits us from seeing how the Relationship between Time Complexity and Size of Collection changes with having

bigger collections for sorting. This argument can be taken as a valid one, as the described experiment does not take into account Collections of much larger size.

Taking this argument into account, we can evaluate that this study can serve as the base idea of conducting further experiments, that will provide insights based on a larger scope of Collection sizes. The same will make possible to study the patterns and changes in the relationship of Time Complexity and number of elements for sorting, within bigger collections.