Exploration and Presentation

Investigating Time Complexity of Sorting Algorithms

May 6, 2021

Authors
Rúni Vedel Niclasen - cph-rn118
Camilla Jenny Valerius Staunstrup - cph-cs340
Asger Thorsboe Lundblad - cph-al217

Abstract

Choosing the right sorting algorithm is important to avoid poor run-time efficiency. Not understanding the effect that the time complexity of an algorithm can have on run-time performance, makes it difficult to choose the proper sorting algorithm for the given problem. Switching to a sorting algorithm with a better time complexity has more of an effect than upgrading hardware. The impact time complexity have on run-time and what factors to take into account is crucial to understand when deciding which algorithm to implement.

Contents

1 Introduction

The inspiration for this paper comes from an assignment in the course Mathematics and Algorithms, where we examined five different sorting algorithms with varying big O time complexity, by sorting the complete works of William Shakespeare. [1]

1.1 Problem description

We are interested in investigating how the run-time of the five sorting algorithms changes on different hardware and whether the difference reflects the time complexity.
In addition to this, we are interested in delving further into the use cases for the different sorting algorithms and examine whether arguments can be made for the use of the algorithms with a higher time complexity.

Problem scope

What impact does time complexity vs. hardware have on the run-time for five chosen sorting algorithms and in what scenarios would it be beneficial to use a sorting algorithm with a greater time complexity?

1.2 Sorting Algorithms

The five different sorting algorithms we will consider in this paper are the same as in the above-mentioned assignment. We will in this section briefly go over the five sorting algorithms to give a basic overview of their structure and time complexity.

Tries - linear time complexity

A trie is a tree data structure used for lexicographic sorting by locating specific keys (strings) in the search tree. The trie contains nodes with references to child nodes or null and each node have only one parent node. Each node will have references to n child nodes, where n is the size of the alphabet for that specific trie. [2] *page 732-745*
To access a key, the trie is traversed depth-first following the references to other child nodes. For searching and inserting, the time complexity for a trie is dependant on characters stored in the trie and the length of the key which gives $O(n * k)$ where n is the size of the alphabet and k is the length of the key. [3]

Merge sort - linearithmic time complexity

Merge sort is a recursive sorting algorithm that divides an array into two halves, sorts the two halves and then merges the results. One of Merge sort's most attractive properties is that it guarantees to sort any array of n items in time proportional to *O(n log(n))*. Its prime disadvantage is that it uses extra memory proportional to n due to the auxiliary arrays. [2] *page 270-282*
Merge sort has a number of implementations and optimizations, where we chose to use the *Top-down* variation, which follows the *divide-and-conquer* paradigm of dividing a large problem into smaller subproblems, solving them and ultimately merging them to solve the problem.

Heap sort - linearithmic time complexity

A heap is a tree-based data structure where all nodes have a value or weight that is less or equal to all it's child nodes (if it is a min-heap, the opposite is true for a max heap). To satisfy this in a complete binary tree, any child node that have no siblings is assigned to the left, only the bottom rightmost node can have one child node and leaves can only be on the deepest level of the tree. [4]

Heap sort takes advantage of the fact that the root always is the smallest (or greatest), removes the root and swap the rightmost leaf to the root. All children will be smaller than the new root, so the nodes swap places until the heap again satisfy the properties described above. [5] Construction of a heap have a time complexity of $O(n)$ and the extraction of the root have a time complexity of *O(n log(n))*. The majority of the run-time is spent in the extraction of the root-phase. [6]

Insertion sort - quadratic time complexity

Insertion sort is a sorting algorithm that is useful for sorting ordered or almost sorted arrays. Taken from the book *Algorithms, 4th Edition*:

> *The algorithm that people often use to sort bridge hands is to consider the cards one at a time, inserting each into its proper place among those already considered (keeping them sorted).* [2] *page 250*

The algorithm iterates through the array starting at index 1, it then compare the value of the current index to the elements to the left of it. If it finds an element that have a smaller value than itself, it moves all elements it compared itself to one position to the right, while moving itself to the position after the smaller element. The time complexity will for the most part be $O(n^2)$, however if its sorting an already sorted array the time complexity will instead be $O(n)$. [2] *page 250-252* [7]

Selection sort - quadratic time complexity

The Selection sort algorithm is one of the most basic sorting algorithm. For each index in the array, starting at index 0, the algorithm finds the smallest value of the array and swaps it with the current index. The starting index is then incremented by one and the process is repeated until the array has been fully sorted. The time complexity will always be $O(n^2)$ as the algorithm use two nested loops. [2] *page 248-249* [7]

Time Complexity Impact on Choice of Algorithms

When deciding which sorting algorithm is the best fit for a given use case, the time complexity needs to be considered. Algorithms with a time complexity of $O(n^2)$ becomes quadratic slower based on the amount of data in the array its sorting. While algorithms with *O(n log(n))* have a linearithmic growth. One of the best time complexity you can get on a sorting algorithm which can be used on any type of data is *O(n log(n))*. To use an algorithm with a better time complexity, some sort of constraints will need to be applied on the data. An example is Tries which can only be used on a data set of strings. [2] *page 186-205*

When choosing between two or more algorithms with the same time complexity, it is necessary to look at how the different algorithms work. An example is Heap sort and Merge sort which both have a time complexity of *O(n log(n))*.
Heap sort is a bit slower than Merge sort, as it requires preprocessing when it builds the heap, before actually sorting. Merge sort doesn't require any preprocessing, but it use additional memory due to the auxiliary arrays. Aside from time efficiency and memory, the stability of an algorithm also should be considered. In a stable algorithm, elements with identical keys keep their order after being sorted. An example could be if two people with the same age was to be sorted based on their age, a stable sorting algorithm would maintain the order of the ages while a non-stable algorithm would not be able to guarantee the same order. Merge sort is a stable algorithm and Heap sort is a non-stable algorithm [8]. So between these two algorithms the consideration should be on whether the run-time or the memory use is more important, while also considering if the order of identical keys is needed.
Regarding the algorithms with a greater time complexity, such as Selection sort and Insertion sort which both have the time complexity of $O(n^2)$ we will look at the data in the following tests an determine if an argument for their use can be made.

2 Methodology

We implemented the algorithms in Java and measured their performance using all of our available machines, including desktops and laptops, which differ in specifications and age, with the oldest being over 6 years old.

2.1 Hardware specs

In the two tables below, all of the relevant specifications for the six machines used in our tests can be seen.

Name	OS	CPU	RAM	JVM
PC1	Windows 10 Pro 64-bit	Intel i7-4770k @ 3.50 GHz 4-core (8)	16 GB DDR3 @ 1600 MHz	Oracle Corporation 11.0.9
PC2	Windows 10 Pro 64-bit	AMD Ryzen 7 3700X @ 3.60 GHz 8-Core	16 GB DDR4 @ 3200 MHz	Oracle Corporation 11.0.9
PC3	Windows 10 Pro 64-bit	Intel i5-4460 @ 3.2 GHz 4-core	8 GB DDR3 @ 1333 MHz	Oracle Corporation 11.0.10

Table 1: Desktop hardware setup used for testing

Name	OS	CPU	RAM	JVM
PC4	Windows 10 Pro 64-bit	Intel i5-6300HQ @ 2.3 GHz 4-core	6 GB DDR4 @ 2133 MHz	Oracle Corporation 11.0.10
PC5	macOS Big Sur 11.3	Apple M1	8GB	HotSpot 23.25-b01
PC6	Windows 10 Home 64-bit	AMD Ryzen 5 3550H, 4-core	16 GB @ 2100 MHz	Oracle Corporation 11.0.9

Table 2: Laptop hardware setup used for testing

For the IDE, all PCs except PC5 used IntelliJ IDEA version 2020.3.2, Build #IU-203.7148.57. PC5 used Visual Studio Code version 1.55.2.

2.2 Measurement

We measured the algorithms by sorting the complete works of William Shakespeare, as presented in a 6MB text file of 121.762 lines or 5.589.848 characters.
Using Javas *System.nanoTime()* we implemented a Stopwatch class and used it to calculate the running time of the sort. [1]
Effectively, the text was loaded into a String array of 930778 elements and the timer was then started, followed by the sorting method using the String array.
We did two types of runs, one where we tested the run-time of each sorting algorithm on the complete data set on all PCs. We ran the $O(n^2)$ algorithms only once, due to time constraints as these algorithms for most of the PCs took several hours to run, while we ran the other algorithms a few times to make sure the measurements were somewhat reliable.
After this first test, we ran tests of all the sorting algorithms on one PC (PC2) but with varying amounts of data. Due to the generally smaller amount of data, the run-time of the algorithms were significantly shorter which allowed us to run them several times in order to improve the consistency of the measurements.

3 Results

The resulting data from the different runs will be presented in tables and all timings will be in milliseconds unless otherwise specified.

3.1 Timings

We measured the run-time of Heap sort, Merge sort and Trie sort an average of ten times and once for Selection sort and Insertion sort for the timings shown below.

	Selection Sort	Insertion Sort	Heap Sort	Merge Sort	Using a Trie
PC1	10,073,374ms	11,625,154ms	1532ms	1095ms	179ms
PC2	6,443,783ms	9,851,730ms	937ms	654ms	87ms
PC3	8,514,312ms	9,816,525ms	1422ms	1067ms	195ms
PC4	7,590,981ms	9,774,093ms	1478ms	1212ms	178ms
PC5	4,775,843ms	6,293,917ms	760ms	527ms	128ms
PC6	12,259,456ms	12,581,059ms	1952ms	1455ms	247ms

Table 3: Timings for each PC

As seen in the table, PC5 is the strongest in four of the five sorts. This isn't the metric we're looking for, but there appears to be a trend in the percentile differences between the various sorts. Below you will see the converted values of Selection Sort and Insertion Sort for easier reading.

	Selection Sort	Insertion Sort
PC1	167.88 min	193.75 min
PC2	107.39 min	164.19 min
PC3	141.90 min	163.60 min
PC4	126.51 min	162.90 min
PC5	79.59 min	104.89 min
PC6	204.32 min	209.68 min

Table 4: Selection and Insertion sort converted to minutes

We wanted more data for making direct comparisons between the five algorithms. This was gathered by running the algorithms multiple times, at varying data sizes, on the fastest desktop PC. All runs on Trie, Heap and Merge sort were done ten times. Runs on Selection and Insertion sort were done twice when the data amount was between 40% and 100% and ten times for the rest of the data amounts.

Algorithm	Run-time in milliseconds on varying length of data on PC2												
Data size	0.1 %	1 %	2 %	3 %	4 %	5 %	10 %	15 %	20 %	25 %	40 %	75 %	100 %
Insertion	6	173	656	1536	2894	4696	20,057	55,796	93,114	161,085	780,994	5,232,485	9,851,730
Selection	7	187	753	1733	3265	5794	26,506	55,744	90,749	153,894	958,245	3,800,047	6,443,783
Heap	2	6	13	34	49	57	81	124	165	190	428	712	937
Merge	2	5	9	13	21	24	67	164	243	277	409	552	654
Trie	17	30	36	38	40	41	46	50	52	55	73	79	87

Table 5: Run-times of algorithms on varying data amounts

In table 5 all run-times of the five sorting algorithms are displayed. 100 % equals 930778 words of data from the Sorting Shakespeare Assignment [1]

4 Analysis

In this section we will dive a bit deeper and analyze the results we collected in the previous section and present the data in diagrams to ease the comparisons.

4.1 Hardware and Time Complexity

In order to be able to answer our initial question we wanted to make sure whether or not hardware have any impact on the overall difference between the three time complexities the algorithms we are working with represent.

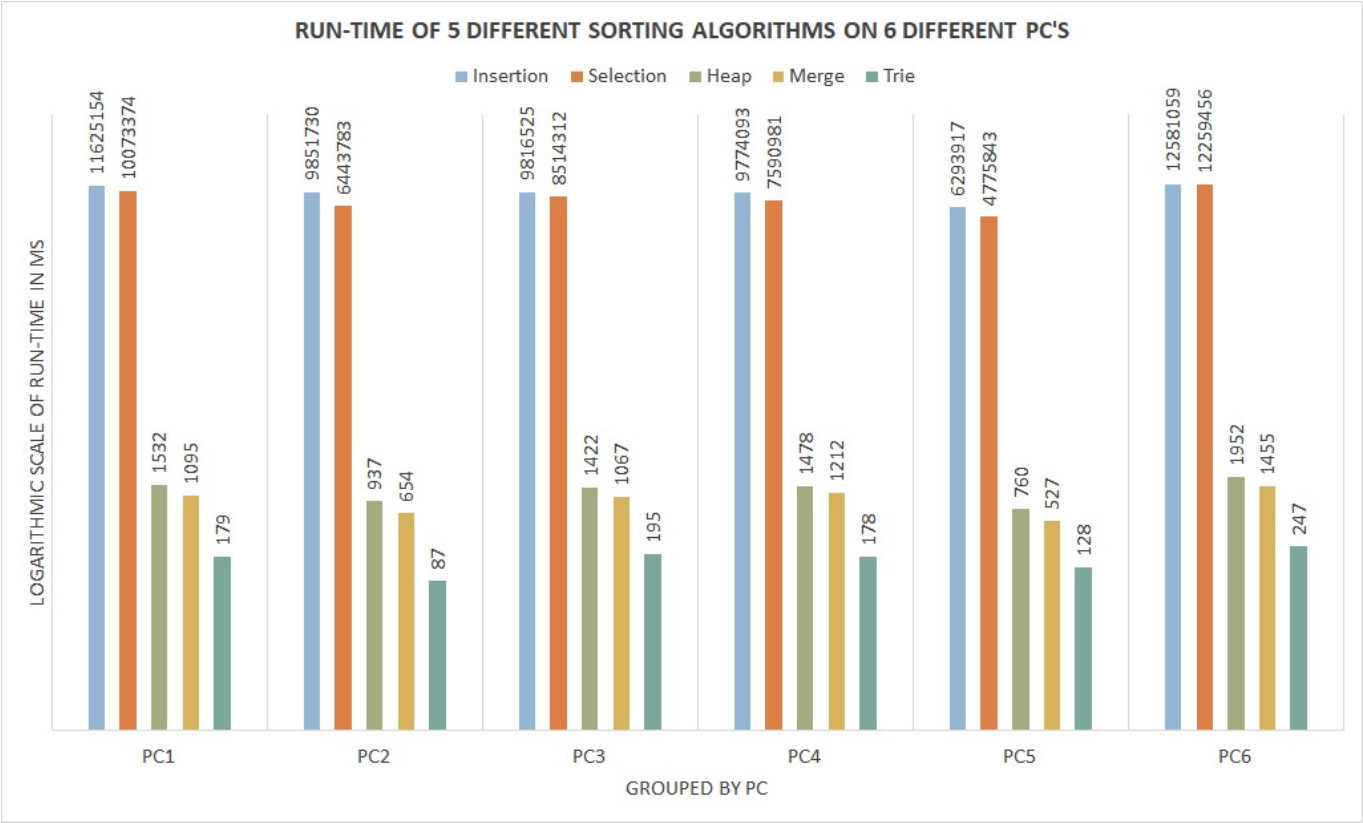

Figure 1: Logarithmic scale representation of the algorithms run-runtime on the six PCs.

In figure 1 above, which is made from the data presented in table 3 on page 4 it is clear to see both the run-time of each sorting algorithm and the apparent jumps between the time complexity for all the PCs.

4.2 Measurements on varying data amounts

As mentioned we decided to run all algorithms on the same PC with varying amounts of data to be able to make more direct comparisons between the five algorithms. The data used to create the following diagrams can be seen in table 5 on page 4.

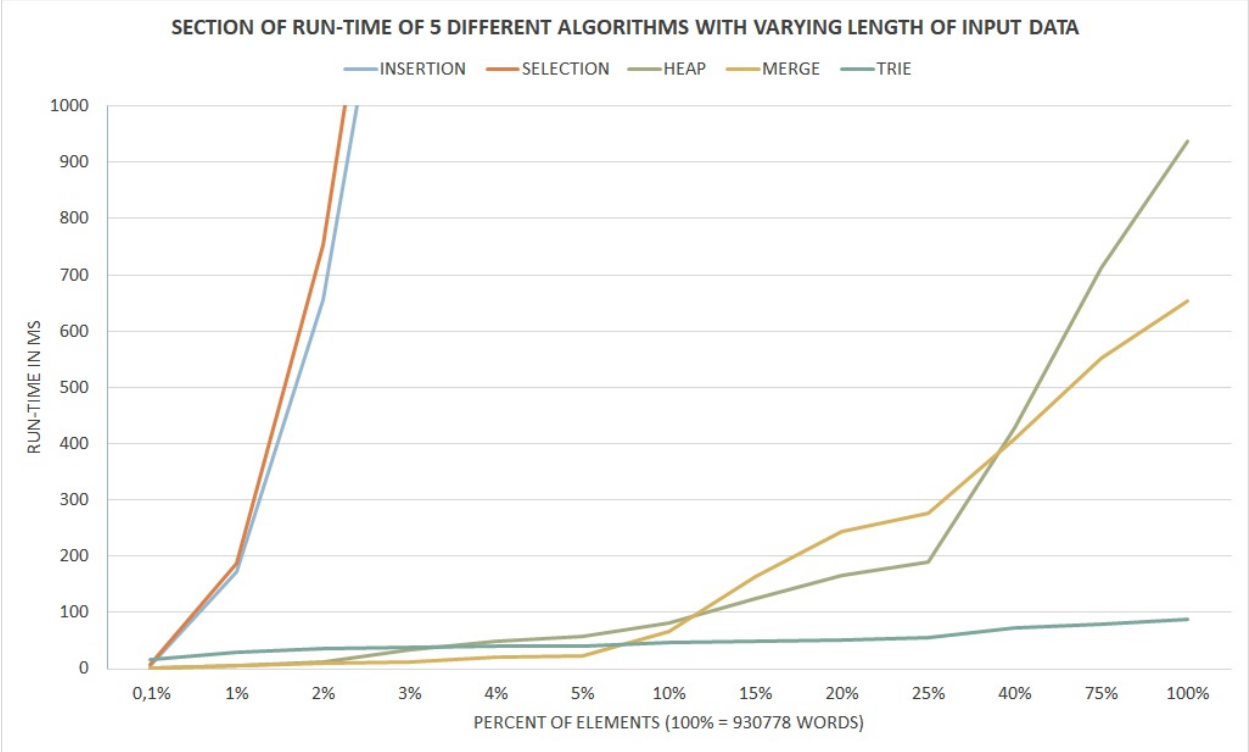

Figure 2: Timings ranging from 2ms to 1000ms. 100% equals 930778 words

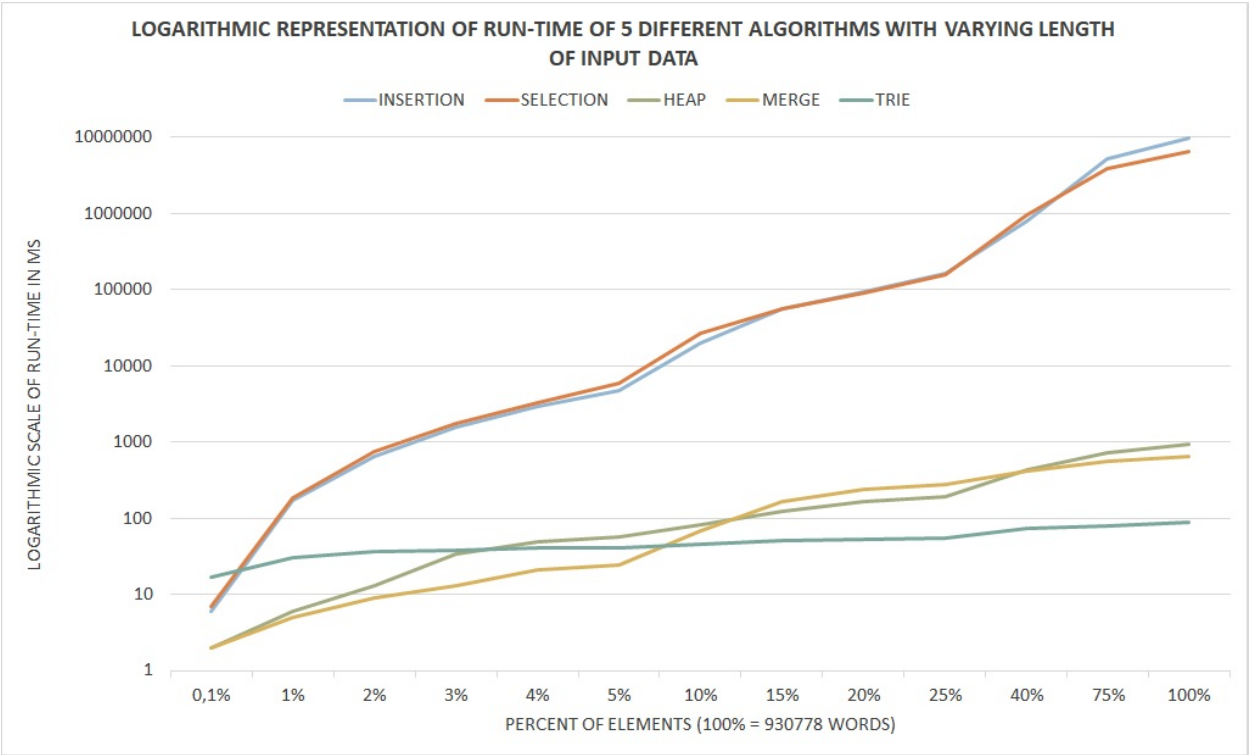

Figure 3: All timings represented on a logarithmic scale. 100% equals 930778 words

Figure 2 shows a smaller section of the timings in order to be able to see all the sorting algorithm graphs clearly, with the varying data amounts. This representation of the data is useful for an overall view of the differences, but in order to compare further we need to look at a logarithmic diagram.
Figure 3 shows all of the timing data on a logarithmic scale with the varying data amounts. On a graph represented on a logarithmic scale, as in figure 3, the distinction between the efficiency for small amounts of data become more clear.

5 Discussion

In this section we aim to compare the findings collected in our two tests with the known time complexity of the sorting algorithms. As well as talking about different use cases for the algorithms with the worst time complexity among the ones we have tested earlier.

5.1 Discussion of results and scope

In figure 1 on page 5 it is quite clear to see that for all the PCs, the relative jump between the time complexities is the same. The hardware does have a small impact on the run-time, but that difference is minuscule compared to the difference between the linear and linearithmic growth and between the linearithmic and quadratic growth. Even if we compare the fastest sorting algorithm on the slowest PC (Trie sort on PC6, run-time of *247ms*), with the next-fastest algorithm on the fastest PC (Merge sort on PC5, run-time of *527ms*), we still see more than a doubling of the run-time from the Trie sort to the Merge sort. The jump from any of the linearithmic algorithms to the quadratic algorithms would clearly never be solved by better hardware.

In figure 2 on page 6 we can see an overall distinction between the three different time complexities and the graphs look similar to graphs shown for educational purposes. [9]. There is a clear difference in efficiency of the algorithms as the input data grows with the quickest and greatest efficiency loss for the to quadratic algorithms. The Trie sort algorithm with linear growth almost seems to loose very little efficiency as the data amount grows.

In figure 3 on page 6 we get a more clear view of the efficiency of the five sorting algorithms compared to each other. The 0.1% of data amounts to 930 words and around this data size all other algorithms are more efficient than the linear time Trie sort. It is interesting that even the quadratic algorithms have a greater efficiency than the Trie when the data amount is this small. This is likely due to the amount of preprocessing in the Trie.

Based on the figures 2 and 3 it is clear that Selection sort and Insertion sort will always have a greater efficiency loss than Merge and Heap sort - as well as Tries after the data set reaches a certain size. However both Selection and Insertion sort have some very useful properties, which can be used in a few different scenarios.
Selection sort is a basic quadratic algorithm which is rarely used. Although it might never be used to sort a whole data set, it's possible to make use of one of its unique properties. If the algorithm is run only N times on an array of data, instead of sorting the whole array, then the first N elements will always be sorted and in their final position based on the final sorted array, while the remaining elements will remain unsorted [8]. As an example suppose Selection sort is run 5 times on this array of integers:

$[5, 2, 7, 12, 9, 11, 1, 1, 9, 34, 7, 0]$

The result would then be:

$[\mathbf{0, 1, 1, 2, 5}, 11, 12, 7, 9, 34, 7, 9]$

The first five entries are sorted and in their correct position, as if the whole array had been sorted. This property can be useful in specific use cases. A use case could for example be a search engine like Google, where an arbitrary search such as "Sorting Algorithms" return 58 million results, but only the 10 most relevant will be shown on page one. In this scenario, running a Selection sort 10 times on the results, rather than sorting the whole result list with any of the other discussed algorithms, might result in a faster loading speed [8].

Insertion sort is another basic quadratic algorithm, however if it's used to sort an already sorted or nearly sorted array of data, then its time complexity becomes linear [2] *page 250*. Because of this it might be possible to combine Insertion sort with a faster algorithm such as Merge or Heap sort. As an example Merge sort could be used to sort the data set to a nearly sorted state and then use insertion sort to finish the sorting, which in theory should speed up the sorting and save some memory use. In addition Insertion sort have a property similar to Selection sort. If it is run N times on an array of data then the first N elements will be sorted, but compared to Selection sort they will not be first N elements of the final sorted array. This can be useful if the use case is sorting of elements which is received from a data stream and they need to be kept sorted in real time [8].

5.2 Reflection

The measurements of the different run-times in our two tests could have been optimized greatly. We have implemented a very basic stopwatch class in Java and we measured the timings from inside the IDE. Furthermore we could have run the sorting algorithms several hundred times instead of the few times they ran.
Practically this was not feasible, even though we would have gotten results that could be used to find standard deviation and mean which would have made outliers apparent, since all computers used for the measurements were our own personal PCs. For a more thorough examination we would follow the advice from Sesoft's paper on *Microbenchmarks in Java and C#*. [10]
Another interesting thing to measure, could be to implement the algorithms in more programming languages beside Java and compare the run time of these implementations.

5.3 Conclusion

The time complexity have a direct impact on the run-time and efficiency of the different algorithms we examined. We saw the same pattern across all PCs in the hardware test. Even with a comparison of Merge sort on the fastest PC and with Trie sort on the slowest PC, showed that the difference in time complexity have a much greater impact on the run-time.

We also noticed that the algorithms with a greater time complexity, such as the quadratic algorithms, are faster than the linear Trie sort if used on a small data set. The quadratic growth first starts to become a greater efficiency loss than the linear growth after a certain data amount. This can be utilized in a use case with a guaranteed small amount of data to achieve greater sorting efficiency.

References

[1] C. Staunstrup, R. Niclasen, and A. Lundblad, "Assignment 3 - sorting shakespeare." www.github.com/Hold-Krykke-BA/MAT-AL/tree/main/Assignment3. Accessed: 2021-04-30.

[2] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[3] F. Pfenning, "Lecture notes on tries, 15-122: Principles of imperative computation." www.cs.cmu.edu/~fp/courses/15122-f10/lectures/18-tries.pdf, 2010. Accessed: 2021-04-30.

[4] D. M. Mount, "University of maryland - cmsc 351 algorithms lecture notes 12." https://www.cs.umd.edu/~meesh/351/mount/lectures/lect12-heaps-and-heapsort.pdf. Accessed: 2021-04-30.

[5] D. M. Mount, "University of maryland - cmsc 351 algorithms lecture notes 13." https://www.cs.umd.edu/~meesh/351/mount/lectures/lect13-heapsort.pdf. Accessed: 2021-04-30.

[6] D. M. Mount, "University of maryland - cmsc 351 algorithms lecture notes 14." https://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf. Accessed: 2021-04-30.

[7] GeeksForGeeks, "Time complexities of all sorting algorithms." https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/. Accessed: 2021-05-04.

[8] P. Mihaylov, "What you don't know about sorting algorithms." https://pmihaylov.com/sorting-algorithms/.

[9] C. Webb, "Complexity and big-o notation in swift." https://medium.com/journey-of-one-thousand-apps/complexity-and-big-o-notation-in-swift-478a67ba20e7. Accessed: 2021-05-01.

[10] P. Sestoft, "Microbenchmarks in java and c#." https://datsoftlyngby.github.io/soft2021spring/resources/ee799a67-SestoftMicrobenchmarking.pdf. Accessed: 2021-05-01; Version 0.8.0 of 2015-09-16.