Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

DA256D, 7,5 credits
Semester Year e.g. Autumn Semester 2021
Faculty of Natural Science

Seminar 1 - Algorithms and data structures

Sandra Kaljula

Author

Sandra Kaljula

Title

Algorithms and data structures – Seminar 1

Supervisor

Kamilla Klonowska

Examiner

Kamilla Klonowska

Abstract

Different algorithms performance can be measured through the average time to complete a few cycles and comparing them to each other. There are many factors that can make the results differ such as the programming language, compiler, how the software is written as well as the hardware specifications. Judging by the calculations and graphs in this report it is very important to know the possible input sizes prior to choosing the best fitting approach for a program. For only small inputs, insertion sort would be the best option. However, for greater inputs it would not be reasonable, because of the bad performance and another algorithm should be chosen instead.

Keywords

Algorithms, data structures, data, sorting, searching, recursive, iterative, big-O.

Content

Introduction

A program that consumes a lot of resources can result in high execution times, memory allocation, usage, leaks, cache performance and etcetera. Profiling is a tool that helps the developer to detect and fix performance bugs causing the excessive resource consumption. Algorithmic profiling specifically analyses the complexity of the program through a series of program runs to get a reliable result. Algorithmic profiling determines a cost function where every input size has its corresponding cost. It also uses a repetition tree(one for each thread) where the focus is on the repetitions(loops and recursions) and inputs[1].

I would personally use algorithmic profilers when developing different sorting and searching algorithms, because it would give me a better understanding on the weak points in the algorithms. This would help me learn from my mistakes and improve the quality of my code. In this report there is a similar comparison made on the following sorting algorithms: quick sort, merge sort, insertion sort and a searching algorithm: binary search.

Research questions:

- Which algorithm is the most efficient?

Limitations:

The algorithms are not guaranteed to be the most efficient ones out there, since they were the first ones found to be working on the internet or the course literature. They are coded and tested using python, which might bring different running times compared to other programming languages.

Method

First, I chose to use python to program my algorithms in, because I have lately only been using Java and I would want to be better at this language too.

Second, a file structure was created with all the files needed for execution. It is very important to have everything organized, since there were over 10 files in total.

Third, the classes were named and documented with doc strings, leaving me with instructions on the next moves. I started with creating the universal wrapper for running all the algorithms, both manually and automatically. When running the program, the user gets to choose between one of them. The manual path takes the user to input the desired number of cycles, input, type of algorithm and eventually a searched value. The last one only being if the user decides for binary search. The automatic path has a set number of cycles and input, which will run on all the algorithms. Lastly the graphs with input size and time will be displayed.

Fourth, I moved into writing the algorithms. It was decided to borrow most of the algorithms from the internet or the course literature and altered them to understand them better as well as adding the partitioning for the quicksort algorithms. The references are found in the references part of the report as well as in the python files.

Lastly, All the algorithms were run automatically on a computer with AMD Ryzen 3 1300X Quad-Core Processor 3.80 GHz. The results were drawn in excel as graphs and tables which are attached with this report. The theoretical expected results were also calculated in excel.

Results

In this part of the report the results retrieved from testing the different algorithms are shown both in tables and graphs. There are also calculations for later discussion the difference between the theoretical and practical outcomes.

Overview

The average time of 5 cycles for selected input sizes between 100 to 1 000 000 for algorithms are shown in figure 1 based on table 1, where (I) stands for iterative and (R) stands for a recursively implemented algorithm. This plot has eleven graphs in it, six of them are solid lines for the algorithms and the five dashed lines are for the most common big-O functions. Some of the values in both plot and the table are clearly missing and the reason behind it will be discussed later in this report.

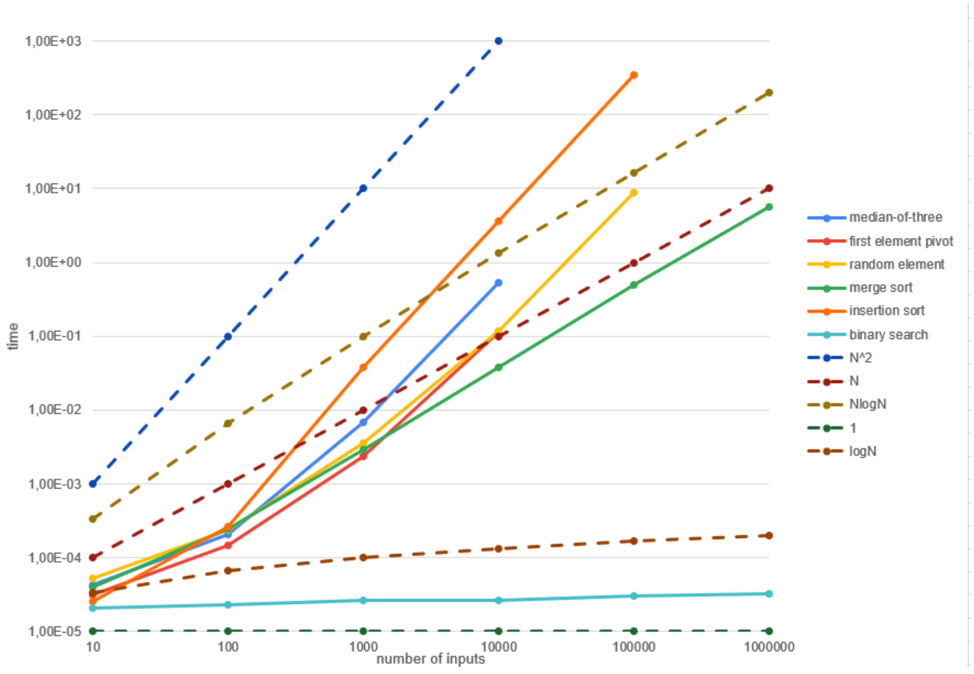

Figure 1 All the algorithms' graphs together with the most common functions for the worst case scenarios.

Algorithm	Size	**10**	**100**	**1 000**	**10 000**	**100 000**	**1000 000**
Quicksort - Median-of-three (R)		0,000042	0,000207	0,006824	0,530199		
Quicksort - First element pivot (R)		0,000031	0,000146	0,002389	0,113950		
Quicksort - Random element pivot (I)		0,000052	0,000241	0,003622	0,116847	8,734018	
Merge sort (R)		0,000039	0,000247	0,002913	0,038337	0,495117	5,702960
Insertion sort (I)		0,000025	0,000262	0,037502	3,682443	347,5476	
Binary search (R)		0,000021	0,000023	0,000026	0,000026	0,000030	0,000032

Table 1 Average time of 5 cycles in seconds in relation to input size.

Quick sort

Quicksort is a fast-sorting algorithm which implementation can be more advanced than the others. The basic idea of quicksort is choosing a value from the given array and comparing it to the other values – sorting them to the left or right of the pivot. After that the two sides will be quick sorted the same way and the process is repeated until a sorted array is retrieved[2].

Time complexity:

Best case: O(N log N)

Average case: O(N log N)

Worst case: $O(N^2)$

Median-of-three (R)		**First element pivot (R)**		**Random element pivot (I)**	
Input size	**Time (s)**	**Input size**	**Time (s)**	**Input size**	**Time (s)**
10	0,000042	10	0,000031	10	0,000052
100	0,000207	100	0,000146	100	0,000241
1000	0,006824	1000	0,002389	1000	0,003622
10000	0,530199	10000	0,113950	10000	0,116847

Table 2 Running time in seconds in relation to the input sizes with the three quicksort algorithms.

More details about different approaches to quicksort are found in the following chapters.

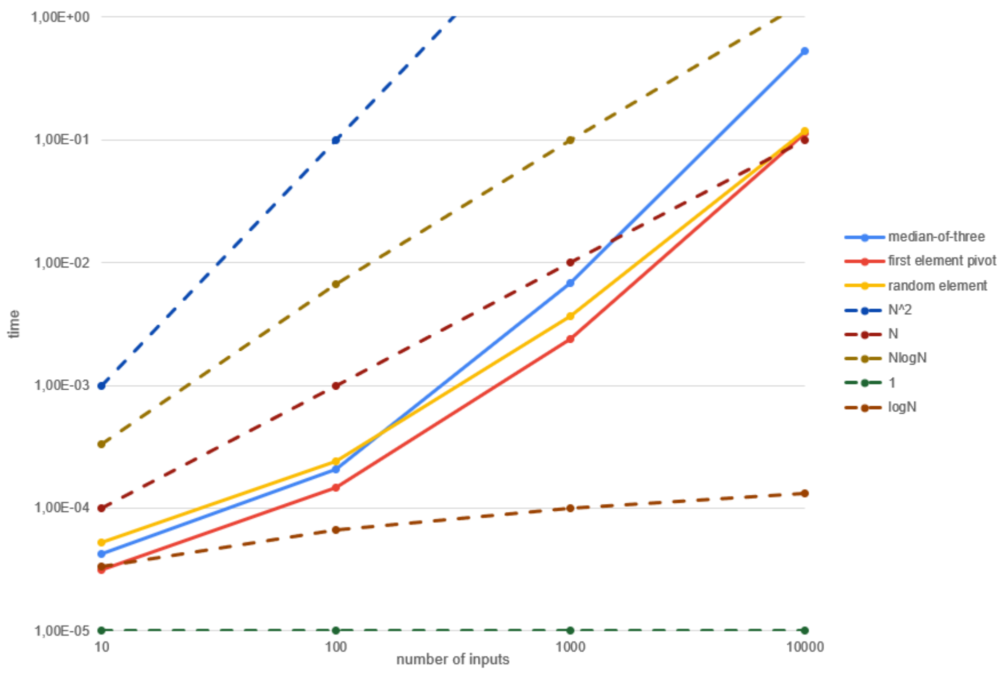

Figure 2 Overview of the three different quicksort algorithms discussed in this report.

Median of three pivot

The median of three pivot partitioning approach of quick sort chooses a pivot by finding the median value of the first, middle and last values in the list. In table 3 there are running times for the selected input sizes.

Input size	Time (s)
10	0,000042
100	0,000207
1000	0,006824
10000	0,530199

Table 3 Running time in seconds in relation to the input sizes with the median-of-three pivot algorithm.

In Figure 3 the graph shows how the median-of-three pivot quick sort algorithm grows in comparison to the most common big-O functions.

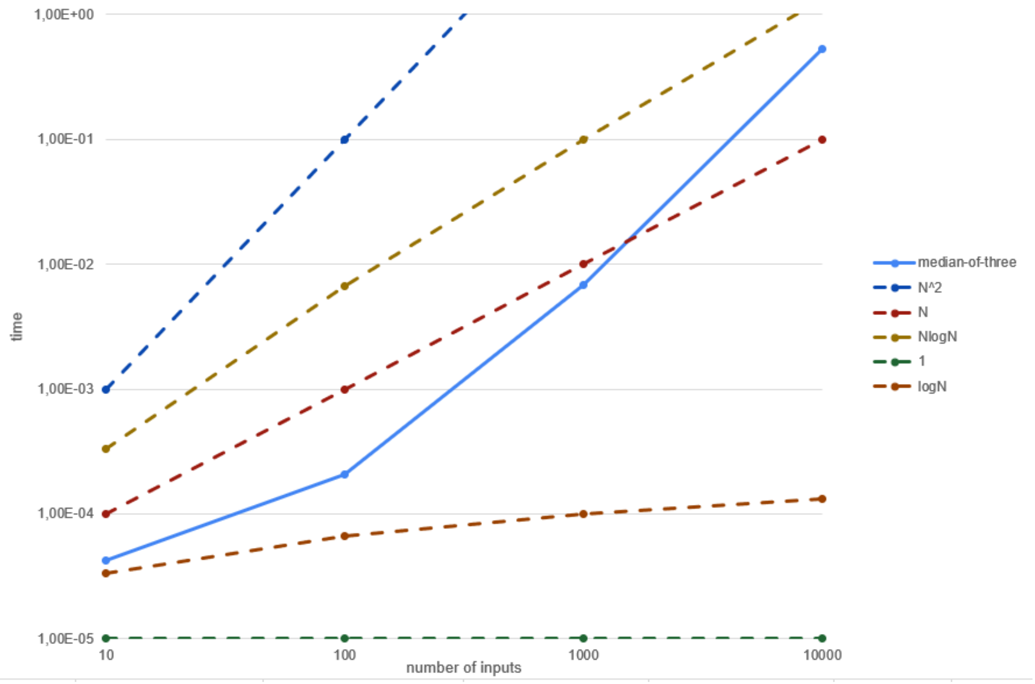

Figure 3 Quicksort - Median-of-three pivot.

Calculations

Average case: O(N log N)

Excel sheet used for a faster calculation Therefore not all the calculations will be shown for .

(1) T(N) = cN log N, T(N) = T(10) = 0,000031 = cN log N

C=$\frac{T(N)}{\mathrm{Nlog}\,N}$= 0,00000126433

T(10 N) = c10N log10N = c10N log N + c10N log10 = T(N) + c10N log10

T(100) = 0,000462

Theoretical T(100) = 0,000462 , Practical T(100) = 0,000207

(2) Theoretical T(1 000) = 0,001242, Practical T(1 000) = 0,006824

(3) Theoretical T(10 000) = 0,029570667, Practical T(10 000) = 0,530199

First array element pivot

The first array/list element pivot partitioning approach of quick sort chooses a pivot by finding the first value in the list. In table 4 there are running times for the selected input sizes.

Input size	Time (s)
10	0,000031
100	0,000146
1000	0,002389
10000	0,113950

Table 4 Running time in relation to the input sizes with the first array element pivot.

In Figure 4 the graph shows how the first array element pivot quick sort algorithm grows in comparison to the most common big-O functions.

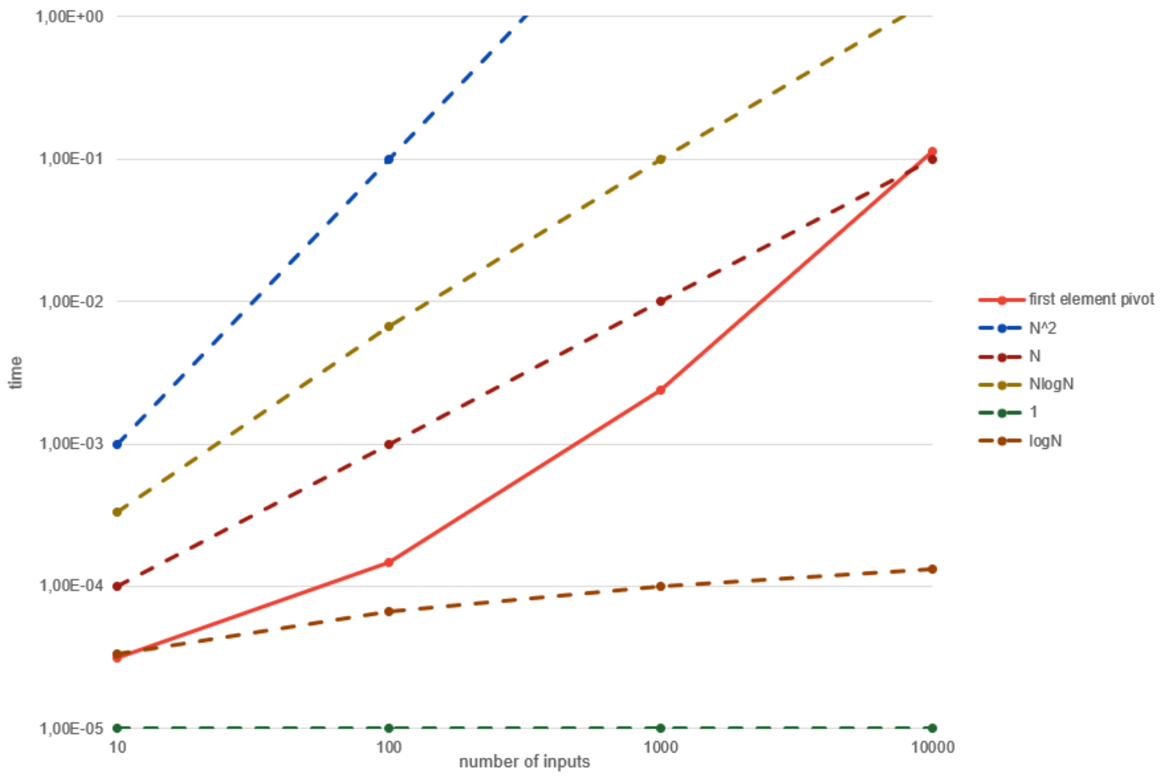

Figure 4 Quicksort - First array element pivot.

Calculations

Average case: O(N log N)

Excel sheet used for a faster calculation Therefore not all the calculations will be shown for .

(1) T(N) = cN log N, T(N) = T(10) = 0,000031 = cN log N

C=$\frac{T(N)}{\mathrm{Nlog}\,N}$= 0,000000933193

T(10 N) = c10N log10N = c10N log N + c10N log10 = T(N) + c10N log10

T(100) =0,000341

Theoretical T(100) =0,000341, Practical T(100) = 0,000146

(2) Theoretical T(1 000) =0,000876 , Practical T(1 000) = 0,002389

(3) Theoretical T(10 000) = 0,010352333, Practical T(10 000) = 0,113950

Random element pivot

The random element pivot partitioning approach of quick sort chooses a pivot by finding a random value from the list. In table 5 there are running times for the selected input sizes.

Input size	Time (s)
10	0,000052
100	0,000241
1000	0,003622
10000	0,116847
100000	8,734018

Table 5 Running time in seconds in relation to the input sizes with the random element pivot algorithm.

In Figure 5 the graph shows how the random element pivot quick sort algorithm grows in comparison to the most common big-O functions.

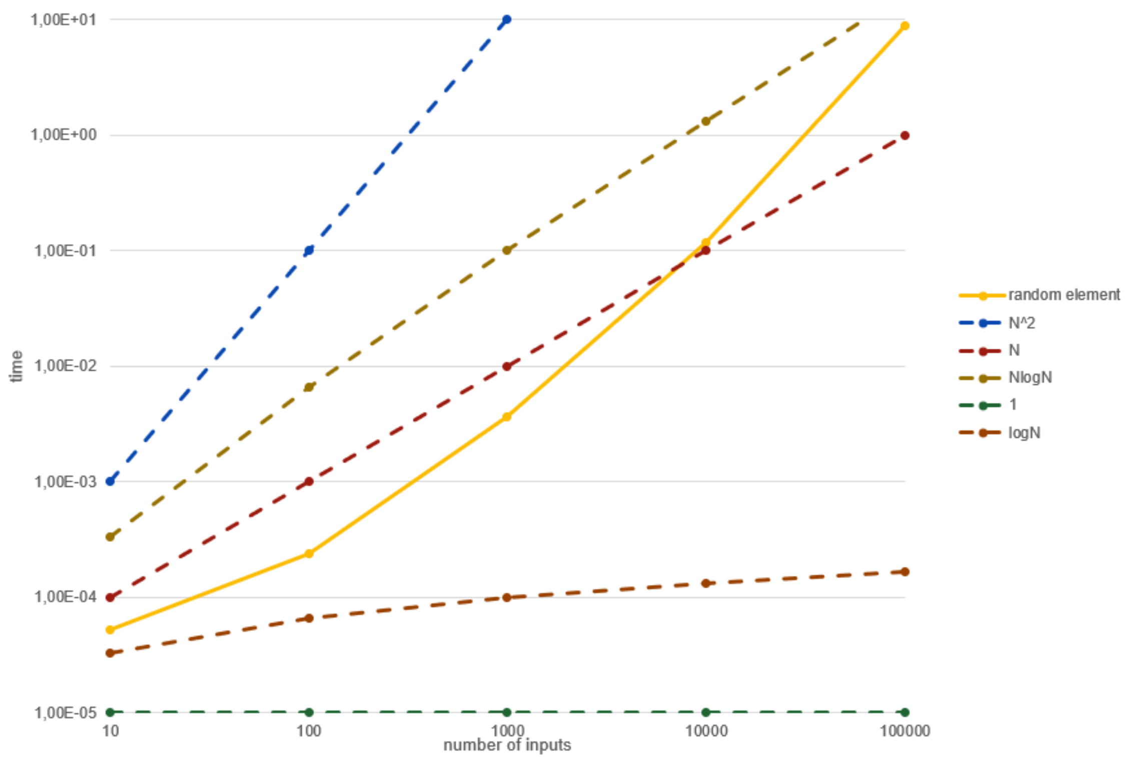

Figure 5 Quicksort - Random element pivot.

Calculations

Average case: O(N log N)

Excel sheet used for a faster calculation Therefore not all the calculations will be shown for .

(1) T(N) = cN log N, T(N) = T(10) = 0,000052 = cN log N

C=$\frac{T(N)}{\text{Nlog}\, N}$= 0,00000156536

T(10 N) = c10N log10N = c10N log N + c10N log10 = T(N) + c10N log10

T(100) = 0,000572

Theoretical T(100) =0,000572, Practical T(100) = 0,000241

(2) Theoretical T(1 000) = 0,001446, Practical T(1 000) = 0,003622

(3) Theoretical T(10 000) = 0,15695333, Practical T(10 000) = 0,116847

(4) Theoretical T(100 000) = 0,4089645, Practical T(100 000) = 8,734018

Merge sort

Merge sort is a sorting algorithm, where an array that needs to be sorted is divided into smaller so-called sub-problems. Then the sub problems are solved by themselves, which often need their own sub-problems. At the end the arrays or solved sub-problems are merged pair by pair until the final solution is retrieved[3]. In table 6 there are running times for the selected input sizes.

Time complexity:

Best case: O(N log N)

Average case: O(N log N)

Worst case: O(N log N)

Input size	**Time (s)**
10	0,000039
100	0,000247
1000	0,002913
10000	0,038337
100000	0,495117
1000000	5,702960

Table 6 Running time in seconds in relation to the input sizes with the merge sort algorithm.

In Figure 6 the graph shows how the merge sort algorithm grows in comparison to the most common big-O functions.

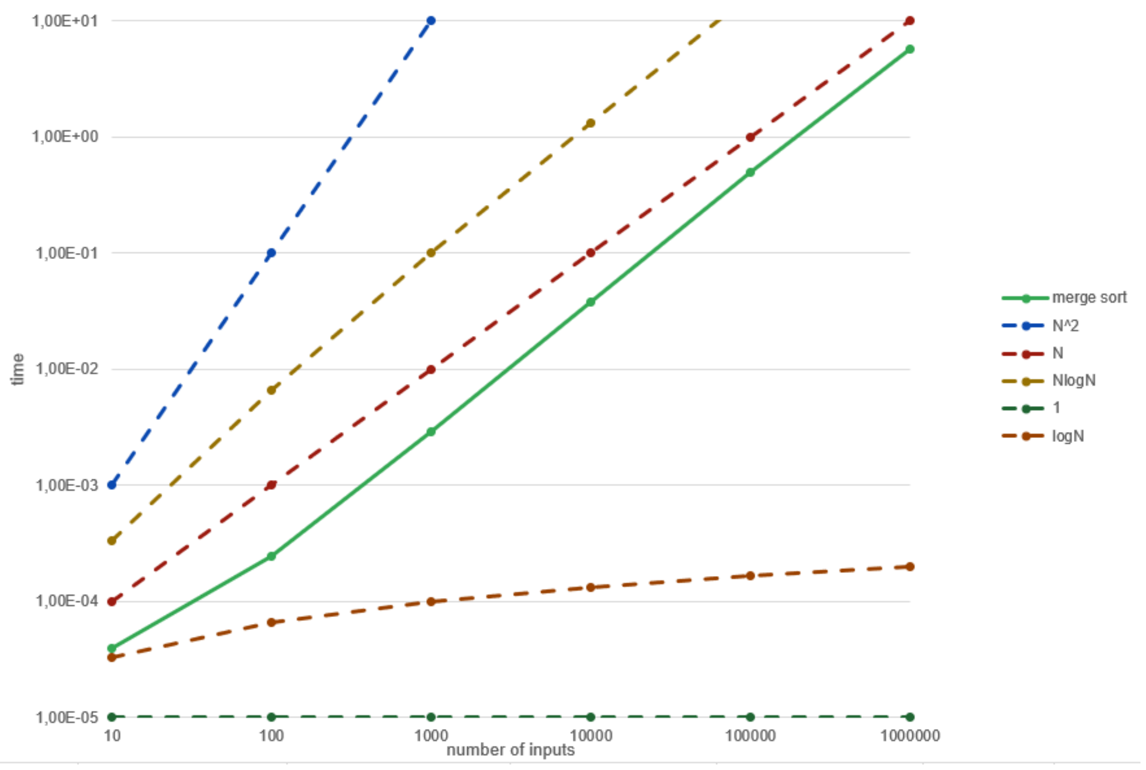

Figure 6 Merge sort.

Calculations

Average case: O(N log N)

Excel sheet used for a faster calculation Therefore not all the calculations will be shown for .

(1) T(N) = cN log N, T(N) = T(10) = 0,000039 = cN log N

C=$\frac{T(N)}{\text{N}\log N}$= 0,00000117402

T(10 N) = c10N log10N = c10N log N + c10N log10 = T(N) + c10N log10

T(100) = 0,000429

Theoretical T(100) = 0,000429, Practical T(100) = 0,000247

(2) Theoretical T(1 000) = 0,001482, Practical T(1 000) = 0,002913

(3) Theoretical T(10 000) =0,012623, Practical T(10 000) = 0,038337

(4) Theoretical T(100 000) = 0,1341795, Practical T(100 000) = 0,495117

(5) Theoretical T(1 000 000) = 1,485351, Practical T(1 000 000) = 5,70296

Insertion sort

Insertion sort is a sorting algorithm that assumes that the first element in the list is already sorted at the first iteration. All the other iterations after that it keeps checking if the key value is less or more than the sorted elements, placing the key element according to that[4]. In table 7 there are running times for the selected input sizes.

Time complexity:

Best case: O(N)

Average case: O(N^2)

Worst case: O(N^2)

Input size	Time (s)
10	0,000025
100	0,000262
1000	0,037502
10000	3,682443
100000	347,5476

Table 7 Running time in seconds in relation to the input sizes with the insertion sort algorithm.

In Figure 7 the graph shows how the insertion sort algorithm grows in comparison to the most common big-O functions.

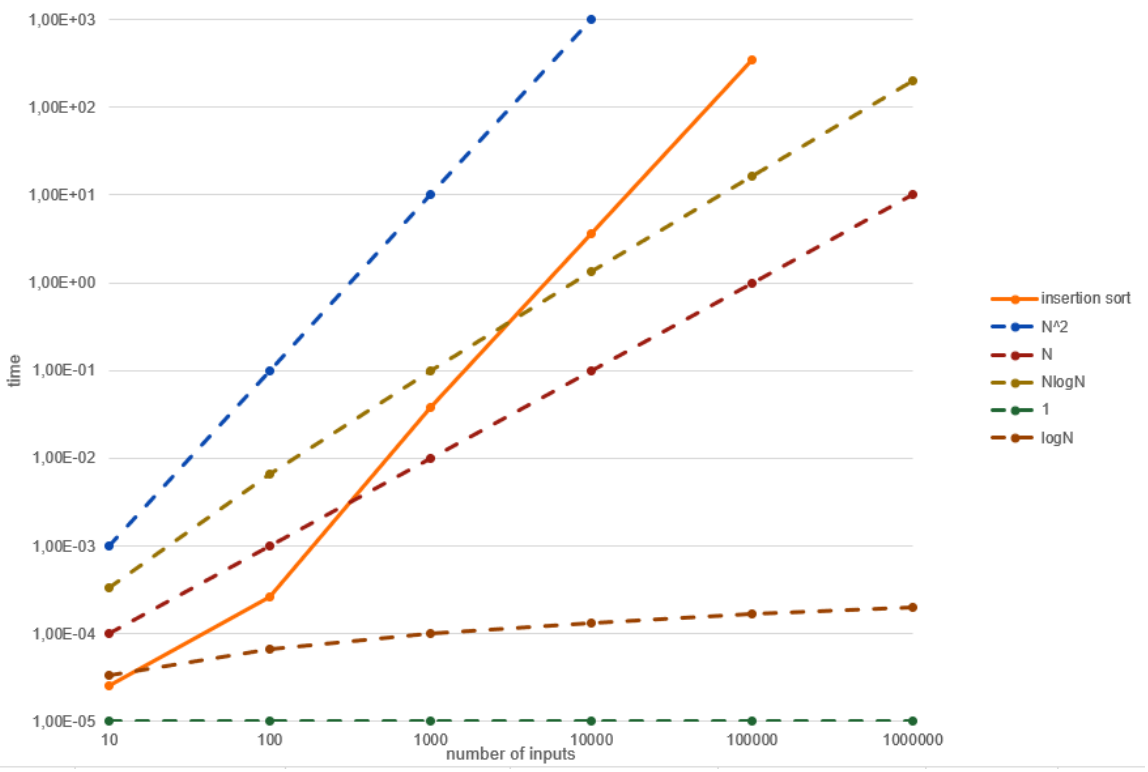

Figure 7 Insertion sort.

Calculations

Average case: $O(N^2)$

(1)

$T(N) = cN^2$, $T(N) = T(10) = 0{,}000025 = cN^2$

$T(10\ N) = c(10N)^2 = cN^2 100 = 0{,}000025*100 = 0{,}0025$

Theoretical T(100) = 0,0025, Practical T(100) = 0,000262

(2)

$T(N) = cN^2$, $T(N) = T(100) = 0{,}000262 = cN^2$

$T(10\ N) = c(10N)^2 = cN^2 100 = 0{,}000262\ *100 = 0{,}0262$

Theoretical T(1 000) = 0,0262, Practical T(1 000) = 0,037502

(3)

$T(N) = cN^2$, $T(N) = T(1\ 000) = 0{,}037502 = cN^2$

$T(10\ N) = c(10N)^2 = cN^2 100 = 0{,}037502*100 = 3{,}7502$

Theoretical T(10 000) = 3,7502, Practical T(10 000) = 3,682443

(4)

T(N) = cN^2, T(N) = T(10 000) = 3,682443 = cN^2

T(10 N) = $c(10N)^2$ = $cN^2 100$ = 3,682443*100 = 368,2443

Theoretical T(100 000) = 368,2443, Practical T(100 000) = 347,5476

(5)

T(N) = cN^2, T(N) = T(100 000) = 347,5476 = cN^2

T(10 N) = $c(10N)^2$ = $cN^2 100$ = 347,5476*100 = 34754,76

Theoretical T(1 000 000) = 34754,76, Practical T(1 000 000) = not tested

(The reason for this is a way too long running times)

Binary search

Binary search is a searching algorithm that only works on a sorted array. Using an unsorted array might cause a faulty result. The algorithm divides the list in two and checks if the middle less, equal or greater than the searched element. If the element is equal to the searched value, the value is found. Otherwise, it selects the side that hopefully has the element and repeats itself[5]. In table 8 there are running times for the selected input sizes.

<u>Time complexity:</u>

Best case: O(1)

Average case: O(log N)

Worst case: O(log N)

Input size	**Time (s)**
10	0,000021
100	0,000023
1000	0,000026
10000	0,000026
100000	0,000030
1000000	0,000032

Table 8 Running time in seconds in relation to the input sizes with the binary search algorithm.

In Figure 8 the graph shows how the binary search algorithm grows in comparison to the most common big-O functions.

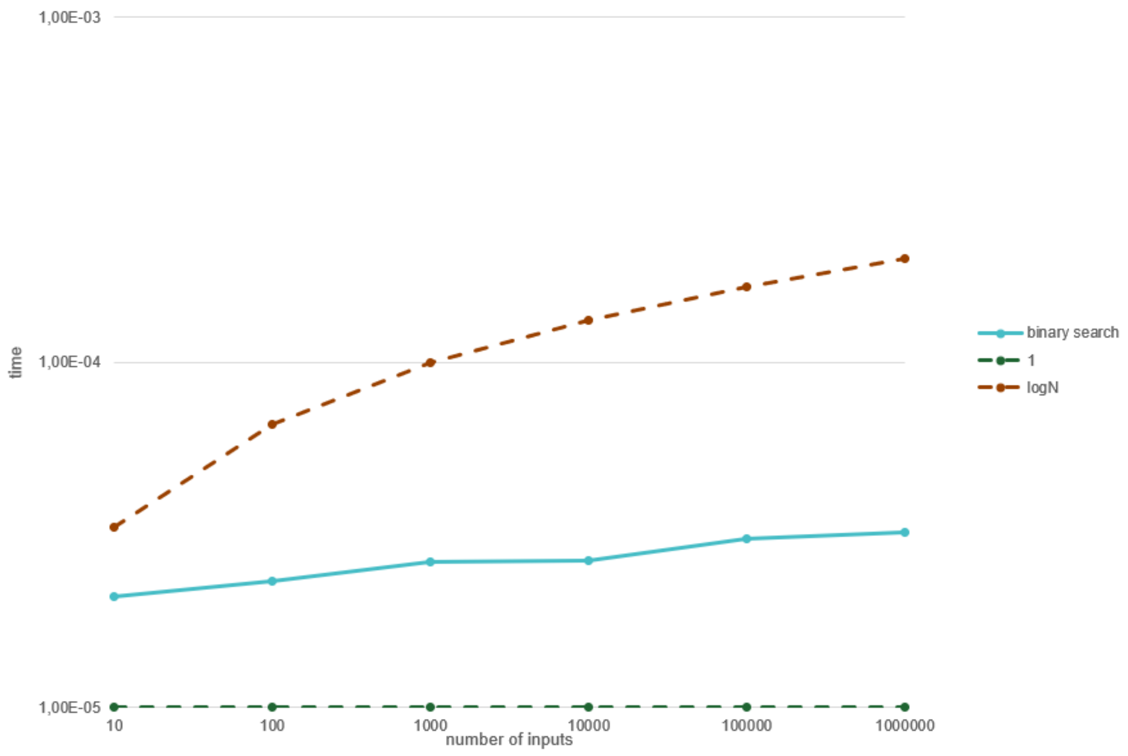

Figure 8 Binary search.

Calculations

Average case: O(log N)

(1)

T(N) = c log N, T(N) = T(10) = 0,000021 = c log N

C=$\frac{T(N)}{\log N}$= 0,00000632162

T(10 N) = c log10N = c log N + c log10 = T(N) + c log10

T(100) = 0,000021 + 0,00000632162*3,32192809489 = 0,00004199996

Theoretical T(100) = 0,00004199996, Practical T(100) = 0,000023

(2)

T(N) = c log N, T(N) = T(100) = 0,000023 = c log N

C=$\frac{T(N)}{\log N}$= 0,00000346184

T(10 N) = c log10N = c log N + c log10 = T(N) + c log10

T(1 000) = 0,000023 + 0,00000346184*3,32192809489 = 0.00003449998

Theoretical T(1 000) = 0,00003449998, Practical T(1 000) = 0,000026

(3)

T(N) = c log N, T(N) = T(1 000) = 0,000026 = c log N

C=$\frac{T(N)}{\log N}$= 0,00000260892

T(10 N) = c log10N = c log N + c log10 = T(N) + c log10

T(10 000) = 0,000026 + 0,00000260892*3,32192809489 = 0,00003466664

Theoretical T(10 000) =0,00003466664, Practical T(10 000) = 0,000026

(4)

T(N) = c log N, T(N) = T(10 000) = 0,000026 = c log N

C=$\frac{T(N)}{\log N}$= 0,00000195669

T(10 N) = c log10N = c log N + c log10 = T(N) + c log10

T(100 000) = 0,000026 + 0,00000195669 *3,32192809489 = 0,00003249998

Theoretical T(100 000) = 0,00003249998, Practical T(100 000) = 0,000030

(5)

T(N) = c log N, T(N) = T(100 000) = 0,000030 = c log N

C=$\frac{T(N)}{\log N}$= 0,00000180617

T(10 N) = c log10N = c log N + c log10 = T(N) + c log10

T(1 000 000) =0,000030 + 0,00000180617*3,32192809489 = 0,00003599996

Theoretical T(1 000 000) =0.00003599996, Practical T(1 000 000) = 0,000032

Analysis

The benchmarking results in this report are the average of 5 cycles. These calculations' part shows the programs performance in relation to theoretical average case scenarios.

First, the quicksort algorithms. Some of which inputs could not be tested for the recursive algorithms because of the recursion error that occurred once there were too many recursive calls made that the environment could not handle. Therefore, the algorithm was only tested to its reachable limit. The quicksort algorithms had a significant difference between the expected and the practical result. For small inputs the algorithms all performed two times faster than expected and towards the end they seemed to be almost have a quadratic difference between the two results – meaning that it was nearing the worst-case scenario the more inputs there were. The results were pretty like each other, but median of three partitioning seemed to get worst performance the more inputs there were. This could be in result of the algorithm not being perfectly written. The code for all the algorithms was borrowed from the internet and some alterations like variable names and changing the partitioning approach were made. Especially for the median of three partitioning there was significantly more lines added to the code than the others, which might be the cause of slower running times. Random element pivot partitioning algorithm was the only algorithm that was implemented iterative, which could also be another factor to why it took the random element pivot to start off slower than the other algorithms and then later evening out to the other algorithms during the larger inputs. The last reason could be that the computer's performance being slowed down during running the testing for that algorithm.

Second, the merge sort algorithm. A similar scenario to what happened to quick search is seen. Although the theoretical average is much more accurate with small inputs than large ones, the merge sort algorithm seemed to be more stable in a long run compared to insertion sort when dealing with larger inputs.

Third, the insertion sort algorithm, which was effective for small inputs. However, when sorting larger inputs, the time seems to nearly the worst-case scenario almost every time. It should not be forgot that the algorithm has both its average and worst-case scenario defined by the same function. The calculations on the practical results show that for some results it even gets slightly higher than the expected, which is even worse than the quadratic growth. As seen in the last calculation the average for one million inputs could not be run for the five cycles as for the other algorithms. The reason behind that was that one cycle of one million inputs would take around 34755 seconds which is around 9.6 hours, meaning that it would have taken at least 48 hours to complete the test run just for this algorithm, therefore it was chosen not to include it in the results.

Fourth and final, the most stable algorithm of them all – binary search. From the calculations perspective both the theoretical and practical results were nearly perfect. There is an obvious difference, this algorithm does not have the same purpose as the other two. This algorithm searches through an already sorted list without having to move anything around as the sorting algorithms do. This is the main reason behind such a drastic difference in performance in relation to other algorithms discussed previously in this report.

Discussion

It might have been interesting to see the results with more input sizes to get a clearer graph of how the performance varies over time.

Also, instead of using the standard way of profiling as we did previously, an algorithmic profiler could have been used during the development process, where more details about the iterations would be provided. The algorithms could have been made more effective, which might make significant differences in the results of the practical outcomes.

These tests are very important for the choice of right algorithm in a program. When choosing an algorithm for a purpose of sorting large inputs all the time insertion search should not be chosen. As noted earlier, it would take at least 48 hours to redeem a result of five cycles of one million inputs, this would not be reasonable or sustainable due to such low performance and the large electricity cost for this approach.

When choosing an algorithm for a purpose of sorting varying sizes of inputs I would personally choose merge sort, because it does not have the worst performance at small inputs and it has the best performance for large inputs, which is very important.

For sorting small inputs and only small inputs, for example if the input is limited to a specific and reasonably small input size I would choose insertion sort, since it is more effective when dealing with small inputs as seen previously.

Even though the binary search algorithm was not compared to any other algorithm of the same purpose I would choose to use the algorithm for searching. This algorithm could be implemented together with one of the sorting algorithms mentioned previously, while keeping in mind the goal input sizes for the program.

Conclusion

To summarize, a similar pattern between the sorting algorithms was seen, the algorithms start off significantly better than the expected average case and after a certain number of inputs they get worse and worse over time. This could be a result of using python and its compiler being much slower than javas.

When choosing an algorithm for sorting, a good decision can be made once the expected input sizes are known.

This project taught me a lot of patience and gave me an insight to software performance testing. I got to learn about different ways sorting and searching can be done and how to make alterations in that kind of code. I also learned to use excel for graphs and more advanced calculations which made the calculations process very efficient.

References

[1] Zaparanuks D, Hauswirth M. Algorithmic profiling. 2012 June; p. 67-69.

https://hkr.instructure.com/courses/4156/files/744349?module_item_id=191056

[2] Codeflow Blekinge. Quick sort [Internet]. Blekinge: C#skolan; 2011[updated date 2011-02-10; cited date 2021-11-20]. Available from: https://csharpskolan.se/article/quick-sort/

[3] Programiz. Merge Sort Algorithm [Internet]. Parewa Labs; [cited date 2021-11-20]. Available from: https://www.programiz.com/dsa/merge-sort

[4] Insertion Sort Algortihm [Internet]. Parewa Labs; [cited date 2021-11-20]. Available from: https://www.programiz.com/dsa/insertion-sort

[5] GeeksforGeeks. Binary Search [Internet]. GeeksforGeeks; [cited date 2021-11-20]. Available from: https://www.geeksforgeeks.org/binary-search/

[6] Weiss, Mark. A. (2012), Data structures and algorithm analysis in Java. 3rd edition Harlow, Essex : Pearson. (632 p).

The code(also mentioned in each python file):

1. GeeksforGeeks. Binary Search [Internet]. GeeksforGeeks; [cited date 2021-11-16]. Available from: https://www.geeksforgeeks.org/binary-search/

2. Python program for insertion sort [Internet]. Parewa Labs; [cited date 2021-11-16]. Available from: https://www.geeksforgeeks.org/python-program-for-insertion-sort/

3. mycodeschool. Quicksort algorithm[cideo file]. 2013, July 23 [cited 2021-11-16]. Available from: https://www.youtube.com/watch?v=COk73cpQbFQ&list=PL2_aWCzGMAwKedT2KfDMB9YA5DgASZb3U&index=9&ab_channel=mycodeschool

4. Webdevjaz. “recursive merge sort python” Code Answer’s[Internet]. CodeGrepper; 2020 [updated date 2020-01-07; cited date 2021-11-16]. Available

from: https://www.codegrepper.com/code-examples/python/recursive+merge+sort+python

5. Studytonight Technologies. Python program for Iterative QuickSort [Internet]. Study Tonight. [cited 2021-11-16]. Available from: https://www.studytonight.com/python-programs/python-program-for-iterative-quicksort

Appendix