A Theoretical and Experimental Comparison of Sorting Algorithms

Maria Fîntîneanu
Department of Computer Science,
Faculty of Mathematics and Computer Science,
Universitatea de Vest Timişoara, România,
Email: `maria.fintineanu04@e-uvt.ro`

June 17, 2023

Abstract

There are numerous fundamental areas in computer science, networks, and artificial intelligence that present various challenges. Sorting algorithms are one of these challenges, requiring researchers to develop algorithms with improved performance. Several factors are considered to achieve better performance, including time complexity, stability, memory space, as well as ease of comprehension and readability. With the ever-increasing need to store vast amounts of data, achieving high efficiency is crucial. Therefore, this paper aims to compare various sorting algorithms, taking into account the aforementioned factors.

Contents

List of Tables

List of Figures

1 Introduction

Sorting algorithms are very useful and it is more useful to know exactly each algorithm for what they are suited the best. Some of these algorithms are more appropriate to implement for large data and efficiency, others are more suitable for saving memory, and others they are just the best algorithms to just understand the basic idea of a sorting algorithm.

1.1 Motivation

Sorting is an important thematic in computer science, because they are the base for various data processing tasks. Even though there are a lot of sorting algorithms, there still is needed for more efficient ones that can handle large data sets with high speed and accuracy. Existing sorting algorithms are very various in their time complexity, stability and memory usage, and how easy it is to implement them, but none of them are perfect to handle all of the factors mentioned.

1.2 Informal description of solution and example

To identify the most optimal sorting algorithm, it is necessary to gather and analyze the most important algorithms that have been implemented until today, and explore the possibility of combining their solutions. In order to achieve this, it is crucial to collect comprehensive data on the performance of each algorithm the time or the memory usage. For example bubble sort and selection sort are a very simple and easy to understand and implement sorting algorithms, but they are inefficient for large data sets, while merge sort and quick sort are more efficient, but may suffer from issues like instability. In this case, this can be done by implementing different sorting algorithms in a programming language and then using different functionalities, like measuring the time to see a result. By collecting the data in a single location it can provide a much better overview. Various methods can be used to achieve this, including tables, histograms, and circle graphs, or any other graph representation, as these visual representations can help in understanding and interpreting the data. By analyzing and comparing the performance of different sorting algorithms it is possible to identify their strengths and weaknesses, and ultimately develop more efficient and effective algorithms.

1.3 Declaration of originality

My work and contribution in this paper is the collection of all the algorithms that will be given later in this paper and it also will mark the big difference between them in tables and graphic representations.

1.4 Reading instructions

The structure of this paper is presented as follows: firstly, it provides and explanation on how the sorting algorithms were implemented, tested, and how the results were displayed in the compiler. Following this, the paper proceeds with the description of the experiments, including tables and graphic representations, and additional observations. The collected information is then compared with related work, which leads to the presentation of conclusions and suggestions for further research on this topic.

2 Model and Implementation of Problem and solution

Details about the source code, where the sorting algorithms are implemented in C++, can be found in the following link.

`https://github.com/MariaF0305/Project1_SortingAlgorithms/blob/main/README.md?plain=1`.

In the current paper we consider the following sorting algorithms, their main idea, how they work and also their pseudo code.

Insertion sort : Insertion sort is a simple sorting algorithm that goes through an array of elements and inserts each element in its proper position in a sorted subarray.

Selection sort: Selection sort is another simple sorting algorithm that works by repeatedly selcting the smallests unsorted element and swapping it with the element at the beginning of the sorted subarray.

Bubble sort: Bubble sort is a simple sorting algorithm that works by repeatedly swapping neighboring elements if they are in the wrong order until the array is sorted.

Merge sort: Merge sort is a divide-and-conquer sorting algorithm that works by recusively dividing the array into two halves, sorting each half, and then mergin the two sorted halves back together.

Quick sort: Quick sort is another divide-and-conquer sorting algorithm that works by selecting a pivot element, partitioning the array around the pivot, and recursively sorting the two resulting subarrays.

Heap sort: Heap sort is a comparison-based sorting algorithm that works by first building a max heap from the array, then repeatedly extracting the maximum element from the heap and putting it at the end of the sorted array.

Count sort: Count sort is a non-comparison-based sorting algorithm that works by counting the number of occurrences of each element in the array, then using these counts to reconstruct the sorted array.

Radix sort: Radix sort is another non-comparison-based sorting algorithm that works by sorting the elements based on their digits, starting from the least significant digit and moving to the most significant digit.

Bucket sort: Bucket sort is a non-comparison-based sorting algorithm that works by dividing the array into a set of buckets, sorting the elements in each bucket, and then putting the buckets back together.

Pancake sort: Pancake sort is a sorting algorithm that works by repeatedly flipping the first k elements of the array to reverse their order until the entire array is sorted.

More about the sorting algorithms you can find in the following sources:
[Estivill-Castro and Wood(1992)]
[Hammad(2015)]
[Taiwo et al.(2020)Taiwo, Christianah, Oluwatobi, Aderonke, et al.]

3 Case studies/Experiment

In this chapter, the experimental results of the sorting algorithms will be presented, which are shown through tables and graphs. Each table is accompanied by observations regarding the differences observed and peculiarities encountered for each algorithm, but after each table are also will be a deeper analysis why each pecularity happen. The sorting algorithms were tested on randomly generated lists, which have the following properties: unsorted, almost sorted, decreasingly sorted, and sorted. The results of the experiment can be observed in the following tables.

Sorting algorithms	Unsorted	Almost sorted	Decreasing sorted	Sorted
Insertion sort	0.0033	0.0031	0.0038	0.0031
Selection sort	0.0062	0.0037	0.0035	0.0041
Bubble sort	0.0037	0.0034	0.0044	0.0027
Merge sort	0.0261	0.0367	0.0192	0.0186
Quick sort	0.0016	0.0017	0.0024	0.0019
Heap sort	0.005	0.0044	0.0043	0.0044
Count sort	0.0071	0.0075	0.0074	0.0071
Radix sort	0.0066	0.0062	0.0091	0.0078
Bucket sort	0.019	0.0207	0.0302	0.0158
Pancake sort	0.004	0.0044	0.0035	0.0043

Table 1: Time in milliseconds sorting 10 elements.

In the initial table, where only ten elements are sorted, it can be observed that even though merge sort is known to be highly efficient for large data, it takes the longest execution time among all the algorithms. This finding is noteworthy, because it is known that the merge algorithm divides the list in two and compares until it reaches the bottom and then recursively return the sorted algorithm. This kind of algorithm is very useful for big data to sort, but for small there much more efficient algorithm that takes half of the time.

In the second table, although the execution time for merge sort remains relatively high, it is no longer the slowest algorithm. Bucket sort has surpassed it in terms of execution time, which is not unexpected given the fact that bucket sort performs operations that are computationally expensive.

As the number of elements increases to one thousand, the efficiency of certain sorting algorithms begins to decline. Pancake sort and bucket sort show lower efficiency, and the performance of insertion sort, selection sort, and bubble sort also deteriorates with larger data. However, statistical analysis indicates that merge sort has become relatively efficient. Meanwhile, quick sort, heap sort, count sort, and radix sort show their strength in handling

Sorting algorithms	Unsorted	Almost sorted	Decreasing sorted	Sorted
Insertion sort	0.0382	0.0184	0.0907	0.0049
Selection sort	0.0413	0.0381	0.186	0.0437
Bubble sort	0.0868	0.0578	0.0203	0.0041
Merge sort	0.1907	0.2664	0.0151	0.1483
Quick sort	0.0185	0.0409	0.0241	0.0874
Heap sort	0.0293	0.0339	0.0349	0.0343
Count sort	0.0133	0.0123	0.1774	0.0103
Radix sort	0.0191	0.0191	0.0916	0.0144
Bucket sort	0.2145	0.1904	0.0427	0.1775
Pancake sort	0.0869	0.1208	0.0658	0.0857

Table 2: Time in milliseconds sorting 100 elements.

Sorting algorithms	Unsorted	Almost sorted	Decreasing sorted	Sorted
Insertion sort	2.6385	0.0334	6.1477	0.0189
Selection sort	2.9647	3.0721	3.1007	3.1741
Bubble sort	6.199	3.0034	8.004	0.0129
Merge sort	1.5879	1.416	1.665	1.9762
Quick sort	0.9953	2.8649	3.8928	6.9284
Heap sort	0.2563	0.2357	0.2478	0.4242
Count sort	0.0336	0.0329	0.0665	0.0595
Radix sort	0.0904	0.0992	0.0903	0.2319
Bucket sort	12.1338	12.4029	13.2834	12.4337
Pancake sort	7.4198	6.8425	6.5529	7.0398

Table 3: Time in milliseconds sorting 1000 elements.

large data, as can be observed from the subsequent tables.

Sorting algorithms	Unsorted	Almost sorted	Decreasing sorted	Sorted
Insertion sort	710.09	0.3182	594.308	0.118
Selection sort	816.256	255.456	252.133	249.688
Bubble sort	1560.63	249.599	716.421	0.0544
Merge sort	314.157	43.5858	135.811	140.312
Quick sort	132.476	566.593	74.5621	682.876
Heap sort	4.0187	3.1499	3.3686	3.0384
Count sort	0.3208	0.6318	0.3184	0.3138
Radix sort	0.8757	0.7185	0.8676	1.5058
Bucket sort	1272.83	1292.85	1258.94	1194.33
Pancake sort	0.8757	673.679	656.476	682.721

Table 4: Time in milliseconds sorting 10000 elements.

For a list of 10000 elements, pancake sort shows more efficiency for larger data, but only for unsorted lists. In other cases, it takes considerably longer compared to other sorting algorithms. Insertion sort remains inefficient for unsorted lists, but it executes very quickly for sorted and almost sorted lists.

Sorting algorithms	Unsorted	Almost sorted	Decreasing sorted	Sorted
Insertion sort	25909.9	2.4154	56122.1	0.9435
Selection sort	26912	24794.6	25888.9	24854.1
Bubble sort	65544	23839.9	73974.5	0.7453
Merge sort	7005.02	7366.29	8499.7	7405.01
Quick sort	6148.95	22854.1	6400.03	69844.5
Heap sort	39.9707	52.0696	37.4567	36.3001
Count sort	3.4859	3.857	3.4693	3.5644
Radix sort	6.9703	9.5867	6.8948	7.8041
Bucket sort	117153	295485	128486	129795
Pancake sort	59720.3	170590	73411.3	66793.4

Table 5: Time in milliseconds sorting 100000 elements.

The table illustrates the efficiency of counting sort and radix sort in comparison to other sorting algorithms. Moreover, heap sort also demonstrates satisfactory performance in terms of time complexity. However, when testing with 100,000 elements, I encountered issues with memory allocation for quick sort. The operations required significant memory resources, which prompted me to adjust certain options in the compiler to ensure the successful execution of quick sort.

Sorting algorithms	Unsorted	Almost sorted	Decreasing sorted	Sorted
Insertion sort	too much	too much	too much	7.8916
Selection sort	too much	too much	too much	too much
Bubble sort	too much	too much	too much	too much
Merge sort	too much	too much	too much	too much
Quick sort	too much	too much	too much	too much
Heap sort	468.865	428.941	438.337	433.808
Count sort	37.5184	34.9579	32.1512	31.2752
Radix sort	74.0392	76.2815	76.3502	77.5049
Bucket sort	too much	too much	too much	too much
Pancake sort	too much	too much	too much	too much

Table 6: Time in milliseconds sorting 1000000 elements.

When attempting to sort one million elements, only a few algorithms could be executed as the others were taking over five minutes. The algorithms that were able to sort the large dataset were heap sort, count sort, and radix sort. Among the three, count sort was found to be the most efficient according to the results presented in the table. It was also surprising to find that insertion sort was able to handle the sorted list of one million elements.

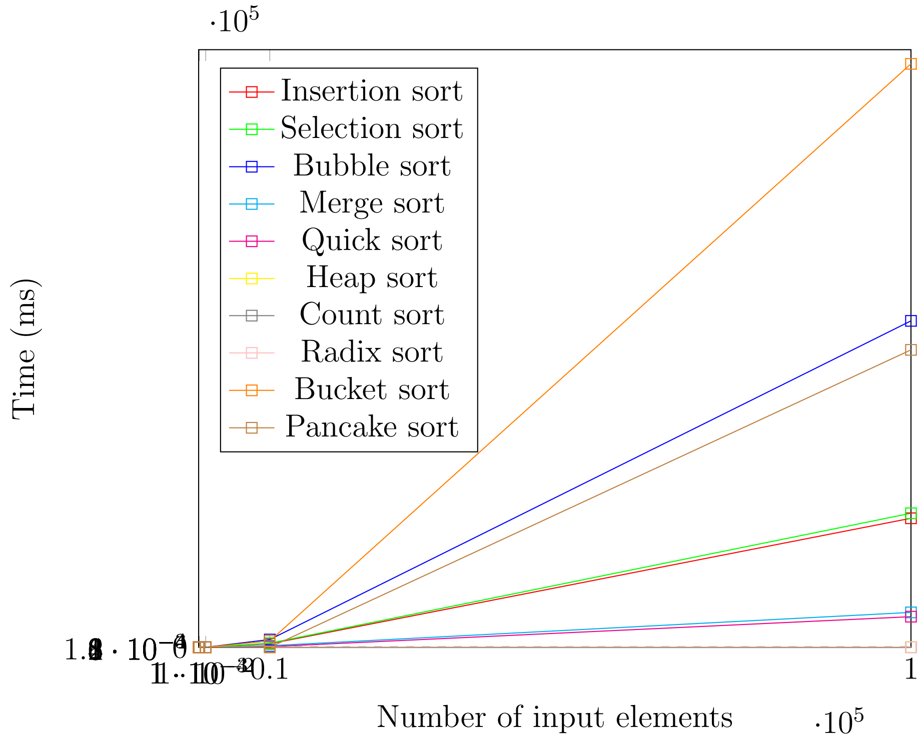

Figure 1: Time behavior of the sorting algorithms when the elements are completely unsorted.

he figure above presents a graphical representation of each sorting algorithm. The data used for the graphs are taken only from the unsorted list and the maximal element in the list is 100000. This choice was made because the first case is considered to be the most relevant.

The graphical representation provides a clearer comparison between the sorting algorithms in terms of time efficiency. From the graph, we can see that bucket sort is the slowest of all the algorithms, while count and radix sort are the fastest. Additionally, it is evident that count and radix sort have similar time efficiency, while insertion and selection sort are also similar in their time efficiency. This is expected because their algorithms are quite similar.

4 Related work

There are several related works to this paper, which focuses on comparing sorting algorithms. One such work is a study that compares various search algorithms based on their performance on different data sets. This study provides valuable insights into the performance of search algorithms, but one disadvantage is that the data sets used in the study may not be representative of all possible scenarios.

Another related work is the comparison of machine learning algorithms, which compares different algorithms based on their accuracy and speed. The advantage of machine learning algorithms is their usefulness in real-world applications, but one disadvantage is that their performance may vary depending on the type of data being analyzed.

A third related work that is relevant to the theme of this paper is the comparison of encryption algorithms, which compares different encryption algorithms based on their security and speed. The advantage of this study is that it helps to select the best encryption algorithm for a given task, but one disadvantage is that the security of encryption algorithms may vary depending on the type of data being encrypted.

Compared to these related works, the study comparing sorting algorithms has the advantage of providing insights into the performance of sorting algorithms on different types of datasets, which can be useful in a variety of applications. However, one disadvantage is that the study only focuses on sorting algorithms and may not be representative of the performance of other types of algorithms. Additionally, the datasets used in the study may not be representative of all possible scenarios.

5 Conclusions and Future Work

In conclusion, the experimental results demonstrate that different sorting algorithms are good for specific use cases, and there is no single algorithm that can be considered the best for all scenarios. Further conclusions drawn from the experiment include the observation that insertion sort is efficient for almost sorted or sorted lists, while insertion, selection, and bubble sort are easy to implement but inefficient for larger lists. Merge sort is not optimal for small lists, taking twice as long as insertion sort, but it performs well overall. Merge and Quick sort are suitable for sorting large lists but are not the most efficient and use a significant amount of memory. Quick sort is particularly inefficient for almost sorted or sorted lists. The best performing sorting algorithms are Heap sort, count sort, pancake sort, and radix sort. Count sort and radix sort are considered more stable as they do not rely on recursive calls like heap sort. On the other hand, bucket sort is the slowest algorithm in all cases: unsorted, almost sorted, decreasing, and sorted.

To further enhance our understanding of sorting algorithms, it would be interesting to determine the exact amount of memory used by each algorithm and compare the results. Additionally, testing sorting algorithms across different programming languages and computing platforms with varying capabilities could yield valuable insights and help us understand the reasons behind any observed differences in performance.

References

[Estivill-Castro and Wood(1992)] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.

[Hammad(2015)] Jehad Hammad. A comparative study between various sorting algorithms. *International Journal of Computer Science and Network Security (IJCSNS)*, 15(3):11, 2015.

[Taiwo et al.(2020)Taiwo, Christianah, Oluwatobi, Aderonke, et al.] Oladipupo Esau Taiwo, Abikoye Oluwakemi Christianah, Akande Noah Oluwatobi, Kayode Anthonia Aderonke, et al. Comparative study of two divide and conquer sorting algorithms: quicksort and mergesort. *Procedia Computer Science*, 171:2532–2540, 2020.