A theoretical and experimental comparison of sorting algorithms

Cătălina Jemna
Student at West University of Timișoara
Email: `catalina.jemna03@e-uvt.ro`

Abstract

This paper analyzes diverse sorting algorithms covered in the Algorithm and Data Structure I class's first semester, exploring them theoretically and experimentally. Theoretical aspects involve time complexity, while practical include their performance across different input sizes. My goal is to provide clear insights into the strengths and weaknesses of these algorithms, facilitating better comprehension.

Contents

1 Introduction

The amount of data is increasing exponentially every day and there is always a need for efficient data organizers, which are sorting algorithms. Traditional sorting algorithms like Bubble sort or Selection sort seem to struggle with the complexity of modern data sets, and that's why the demand for more sophisticated sorting algorithms becomes real.
In this context, it is crucial to understand the strengths and weaknesses of different sorting techniques, to approach specific requirements not only to improve the efficiency of data processing but also to enhance the overall user experience.

A solid understanding of fundamental concepts in computer science, typically acquired during the first year of study, is necessary for a thorough comprehension.

The contribution of this paper lies in a comprehensive analysis of sorting algorithms in a theoretical and practical way. The comparison aims to enhance understanding and decision-making in the field of algorithm optimization.

This paper is organized into sections that dive into different aspects of sorting algorithms. We begin with an introduction to the problem and motivation behind the study. Subsequent sections delve into the methodology, results, discussion, and conclusion, offering an exploration of the following algorithms: Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Heap Sort, Radix Sort, and Merge Sort.

2 Formal Description of Problem and Solution

The problem addressed in this paper is to make a comparison of sorting algorithms across various case scenarios. Selecting an appropriate sorting algorithm based on specific requirements is important because it influences the efficiency and effectiveness of data-processing tasks. To address this problem, the solution involves performing an experiment to evaluate the performance of different sorting algorithms. By examining time complexity, we aim to understand their strengths and weaknesses.
The table below provides a comprehensive overview of the time and space complexity of the algorithms that will be analyzed in this paper:

Algorithms	**Best Case**	**Worst Case**	**Average Case**	**Space Complexity**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Radix Sort	$O(d(n+k))$	$O(d(n+k))$	$O(d(n+k))$	$O(n+k)$

.

Table 1: Complexity of sorting algorithms.

The color scheme in the table above is used to help with the comparison. Red represents the worst algorithm $O(n^2)$. Orange represents the middle algorithm $O(n \log n)$, yellow represents one of the best algorithms$O(d(n+k))$), and the best time complexity is $O(n)$, which is the fastest algorithm, marked by the color green.

Overall, the analysis suggests that:
- Bubble Sort, Insertion Sort, and Selection Sort have poor time complexity in both worst and average-case scenarios, making them less suitable for large datasets;
- Merge Sort, Quick Sort, and Heap Sort show better time complexity, especially in average-case scenarios, making them more efficient choices for sorting large datasets;
- Radix Sort demonstrates linear time complexity, $O(d(n+k))$, where d is the number of digits and k is the range of values, making it efficient for sorting integers with a limited range of values.

3 Model and Implementation of Problem and Solution

System Manual

To model the problem of comparing sorting algorithms on a computer, we utilize C++ as our programming language. Each sorting algorithm is implemented in a separate file, available on GitHub here. Additionally, on the same repository, there is a program that generates different datasets. It will generate random numbers, almost sorted numbers and reverse-sorted numbers. The generated elements are written to a text file, from which they are retrieved and analyzed by all sorting algorithms.

The performance of the algorithms will be analyzed based on 5 sizes of datasets: 10,100,1000,10000, and 100000. Thus, each sorting algorithm will need to process datasets of 5 different sizes and 3 different kinds. The execution time of these will be generated automatically in a CSV file.

User Manual

1. **Accessing Algorithms:** Each sorting algorithm is implemented as a separate file in the GitHub repository, algorithms folder. Users can clone the repository and access individual algorithm implementations as needed.

2. **Generating Input Data:** There is a collection of datasets available in the datsets folder. Within this folder, data is organized into files based on both their size and characteristics. Five distinct datasets are generated, each varying in size: 10,100,1000,10000, and 100000. Each dataset has elements generated randomly, almost sorted, and in reverse-sorted order. These datasets are saved as text files for evaluating and comparing the efficiency of sorting algorithms.

3. **Analyzing Performance:** The performance of each sorting algorithm can be analyzed using the generated input datasets. The execution time of each algorithm is automatically recorded during runtime.

4. **Output Format:** The output of the program is generated and stored into a CSV format file. Therefore, for every algorithm, we'll find the type of dataset used, the number of elements sorted, and the execution time. This file is located in the same directory as the project and can be used for further comparison of sorting algorithms.

The **testing environment** used is a system with a CPU frequency of 2.40GHz, 4 cores, 8 logical processors, and 8GB of RAM at a refresh rate of 60Hz. On the software side, it's a 64-bit operating system, and the code is being run in CodeBlocks 20.03.

4 Case Studies/Experiment

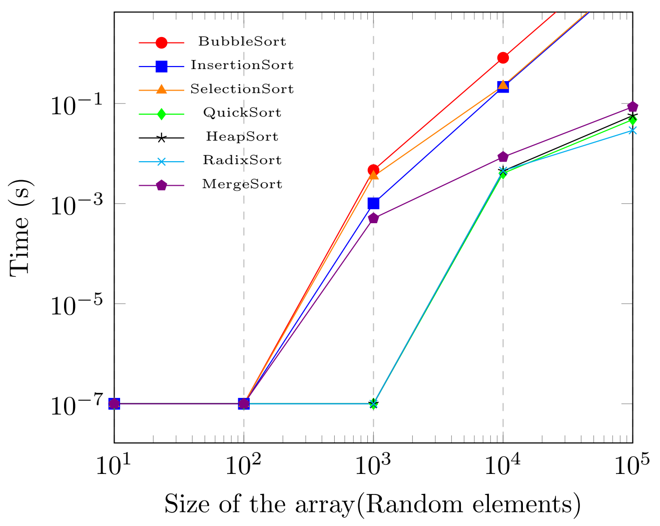
After familiarizing ourselves with the algorithms and their complexity, let's delve into a manual analysis across various case scenarios.

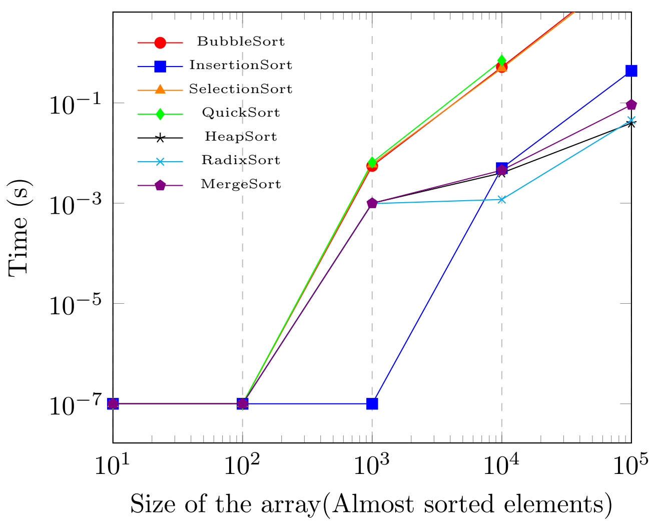
Below, we can observe three graphics illustrating the execution time of each sorting algorithm on various types of elements to be sorted. Each algorithm utilized exactly the same datasets in order to ensure a fair comparison.

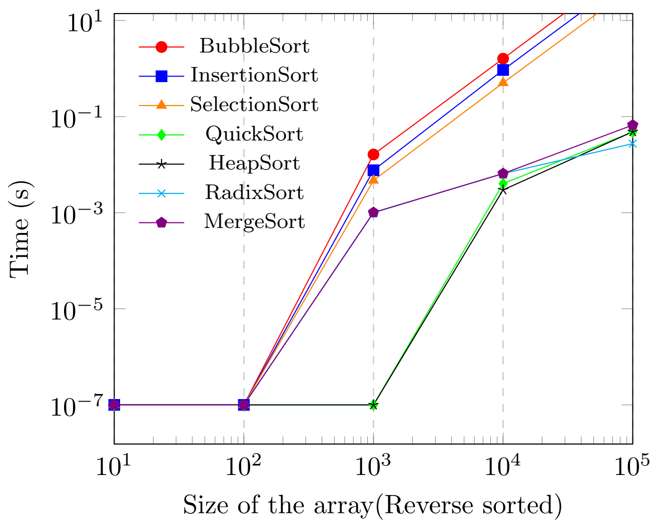
As we observed earlier, when considering time complexity, sorting algorithms such as Merge Sort, Quick Sort, or Heap Sort are expected to perform well, especially for moderate-sized arrays due to their average-case efficiency(Table 1). However, for larger datasets, more complex algorithms like Radix Sort might outperform them in terms of execution time.

All three charts highlight that Bubble Sort, Insertion Sort, and Selection Sort are generally inefficient for large datasets, while Quick Sort, Heap Sort, Radix Sort, and Merge Sort offer better performance, especially as the array size increases. Among these, Radix Sort and Merge Sort appear to be more consistent in their performance across all types of input arrays.

Further,we will delve into each individual scenario in detail.







4.1 Random Elements

Here we have a concrete perspective on how algorithms behave with random elements:

Algorithm	10	100	1000	10000	100000
Bubble Sort	0.0000001	0.0000001	0.004659	0.818711	143.582895
Insertion Sort	0.0000001	0.0000001	0.001010	0.213875	38.104295
Selection Sort	0.0000001	0.0000001	0.003519	0.225197	40.646810
Quick Sort	0.0000001	0.0000001	0.0000001	0.004026	0.047716
Heap Sort	0.0000001	0.0000001	0.0000001	0.004479	0.057232
Radix Sort	0.0000001	0.0000001	0.0000001	0.004535	0.029077
Merge Sort	0.0000001	0.0000001	0.000507	0.008525	0.085711

.

Table 2: Execution Time (s) for Random Elements

Based on the information provided in the table, we can observe that:
- Bubble Sort demonstrates slower performance as the number of elements increases. Therefore, it is not suitable for large datasets;
- Insertion Sort shows a better performance compared to Bubble Sort but still has slower execution times as the dataset grows;
- Selection Sort performs better than Bubble Sort but still shows slower execution times with larger datasets;
- Quick Sort performs well across various dataset sizes, showing consistent execution times, especially for larger datasets;
- Heap Sort shows consistent performance across different dataset sizes, indicating its suitability for various scenarios;
- Radix Sort demonstrates excellent performance, with consistent and relatively fast execution times across different dataset sizes;

- Merge Sort shows stable performance across different dataset sizes, making it a reliable choice for sorting.

After observing how algorithms behave on different datasets we can rank them differently based on the dataset size. For smaller datasets, certain algorithms stand out, while for larger datasets, the ranking might shift.

Rank	**Algorithm**
1	Quick Sort
2	Radix Sort
3	Merge Sort
4	Heap Sort
5	Bubble Sort
6	Selection Sort
7	Insertion Sort

Table 3: Ranking of Sorting Algorithms based on Execution Time for small datasets of random elements

Rank	**Algorithm**
1	Radix Sort
2	Merge Sort
3	Quick Sort
4	Heap Sort
5	Bubble Sort
6	Selection Sort
7	Insertion Sort

Table 4: Ranking of Sorting Algorithms based on Execution Time for large datasets of random elements

In summary, Radix Sort appears to be the standout performer in terms of fast execution across large datasets, while for smaller datasets, Quick Sort takes the lead, closely followed by Radix Sort. As for the remaining algorithms the rankings remain the same across both small and large datasets.

4.2 Almost Sorted Elements

Let's see if the results remain consistent for almost sorted elements.

Table 5: Execution Time (s) for Almost Sorted Elements

Algorithm	10	100	1000	10000	100000
Bubble Sort	0.0000001	0.0000001	0.005532	0.520522	54.074308
Insertion Sort	0.0000001	0.0000001	0.0000001	0.004996	0.436163
Selection Sort	0.0000001	0.0000001	0.005977	0.491278	48.821774
Quick Sort	0.0000001	0.0000001	0.006526	0.702521	0.059634
Heap Sort	0.0000001	0.0000001	0.001000	0.004006	0.039534
Radix Sort	0.0000001	0.0000001	0.000982	0.001186	0.045300
Merge Sort	0.0000001	0.0000001	0.000995	0.004550	0.091603

Based on this table, we can observe the following:
- Bubble Sort shows a notable decrease in execution time compared to random elements;
- Insertion Sort maintains its relatively efficient performance, especially for smaller datasets, demonstrating its adaptability to almost sorted elements;
- Selection Sort shows a notable increase in execution time compared to random elements, especially as the dataset size increases. This is expected since almost sorted elements still require multiple passes through the array for sorting, resulting in higher time complexity;
- Quick Sort still displays competitive performance, but we can observe a increase in execution time compared to random elements;
- Heap Sort maintains consistent performance across different dataset sizes, indicating its robustness even for almost sorted elements;
- Radix Sort and Merge Sort also demonstrate relatively stable execution times, this means that they are not affected by the almost sorted nature of the elements.

Rank	**Algorithm**
1	Quick Sort
2	Heap Sort
3	Radix Sort
4	Merge Sort
5	Insertion Sort
6	Selection Sort
7	Bubble Sort

Table 6: Ranking of Sorting Algorithms based on Execution Time for small datasets of almost sorted elements

Rank	**Algorithm**
1	Radix Sort
2	Quick Sort
3	Heap Sort
4	Merge Sort
5	Insertion Sort
6	Selection Sort
7	Bubble Sort

Table 7: Ranking of Sorting Algorithms based on Execution Time for large datasets of almost sorted elements

Comparing the rankings, we can observe differences between the rankings for small datasets of random elements and small datasets of almost sorted elements. In the ranking for small datasets of random elements, Quick Sort takes the lead, followed by Radix Sort, while in the ranking for small datasets of almost sorted elements, Quick Sort also leads but is followed by Heap Sort. This indicates that for almost sorted elements, Heap Sort performs better than Radix Sort.

Similarly, when comparing the rankings for large datasets of random elements and large datasets of almost sorted elements, we find that Radix Sort takes the lead in both cases. In the ranking for large datasets of random elements, Merge Sort comes second, followed by Quick Sort, while in the ranking for large datasets of almost sorted elements, Quick Sort takes the second position, followed by Heap Sort and then Merge Sort.

4.3 Reverse Sorted Elements

Finally, we have the chance to observe how the algorithms perform with elements sorted in reverse order.

Table 8: Execution Time (s) for Reverse Sorted Elements

Algorithm	10	100	1000	10000	100000
Bubble Sort	0.0000001	0.0000001	0.016253	1.604109	161.068472
Insertion Sort	0.0000001	0.0000001	0.007632	0.932636	92.031155
Selection Sort	0.0000001	0.0000001	0.004689	0.501973	48.894349
Quick Sort	0.0000001	0.0000001	0.0000001	0.004506	0.045634
Heap Sort	0.0000001	0.0000001	0.0000001	0.002979	0.048331
Radix Sort	0.0000001	0.0000001	0.000999	0.006515	0.027326
Merge Sort	0.0000001	0.0000001	0.001009	0.006503	0.066525

Based on the table provided for reverse sorted elements, we can make the following observations about the performance of each sorting algorithm:

- Bubble Sort shows a significant increase in execution time as the dataset size increases. It has the highest execution time among all algorithms, especially for larger datasets;
- Insertion Sort also experiences an increase in execution time with larger dataset sizes but performs better than Bubble Sort. However, its execution time is still relatively high for larger datasets compared to more efficient algorithms;
- Selection Sort is similar to Insertion Sort, it experiences an increase in execution time with larger dataset sizes but performs better than Bubble Sort. However, it's still not as efficient as other sorting algorithms;
- Quick Sort shows a consistent and low execution time across different dataset sizes, making it one of the fastest algorithms for reverse-sorted elements. Its performance is competitive, especially for larger datasets;
- Heap Sort offers a low and consistent execution time across different dataset sizes, similar to Quick Sort. It performs efficiently for reverse-sorted elements, especially for larger datasets;
- Radix Sort shows a relatively low execution time across different dataset sizes, making it one of the faster algorithms for reverse sorted elements. Its performance remains consistent as the dataset size increases;

- Merge Sort demonstrates a moderate increase in execution time with larger dataset sizes but remains relatively low compared to other algorithms. It performs efficiently for reverse-sorted elements.

Rank	**Algorithm**
1	Heap Sort
2	Quick Sort
3	Merge Sort
4	Radix Sort
5	Selection Sort
6	Insertion Sort
7	Bubble Sort

Table 9: Ranking of Sorting Algorithms based on Execution Time for small datasets of reverse sorted elements

Rank	**Algorithm**
1	Radix Sort
2	Quick Sort
3	Heap Sort
4	Merge Sort
5	Selection Sort
6	Insertion Sort
7	Bubble Sort

Table 10: Ranking of Sorting Algorithms based on Execution Time for large datasets of reverse sorted elements

Comparing the rankings, we can note discrepancies between the rankings for small datasets of random elements and small datasets of almost sorted elements. In the ranking for small datasets of random elements, Heap Sort is at the top, with Quick Sort following closely behind. However, in the ranking for small datasets of almost sorted elements, Quick Sort takes the lead, followed by Heap Sort. This suggests that Quick Sort performs better than Heap Sort for almost sorted elements. Specifically, for smaller datasets, the top-performing algorithms are Quick Sort and Heap Sort.
Similarly, when comparing the rankings for large datasets of random elements and large datasets of almost sorted elements, we find that Radix Sort is consistently in the lead in both cases, and the rankings remain unchanged. This suggests that, for larger datasets, Radix Sort consistently outperforms other algorithms.

These differences suggest that the performance of sorting algorithms can vary depending on the characteristics of the dataset. While some algorithms like Radix Sort consistently perform well across different dataset types, others may show variations in their performance rankings based on dataset characteristics such as randomness or pre-sortedness. For smaller datasets, Quick Sort and Heap Sort emerge as the top performers, while for larger datasets, Radix Sort maintains its position as the top-performing algorithm across all cases.

5 Related Work

One related paper, "Sorting Algorithms – A Comparative Study" by Naeem Akhter, Muhammad Idrees, and Furqan-ur-Rehman**(here)**, delves into a detailed analysis of various sorting algorithms, comparing their efficiency based on time complexity. While their study explores a wide range of datasets, I chose to experiment with fewer datasets, each containing a different number of elements. This choice highlights the differences in efficiency more prominently. By utilizing the same datasets for every algorithm,it was provided a clearer comparison of efficiency between the algorithms. Aditionally, I appreciated the concept of including the ranking table for each scenario, and I integrated it into my paper.

Another relevant paper, authored by Khalid Suleiman Al-Kharabsheh, Ibrahim Mahmoud AlTurani, Abdallah Mahmoud Ibrahim AlTurani, and Nabeel Imhammed Zanoon, titled "Review on Sorting Algorithms, A Comparative Study," **(here)** significantly contributed to the development of the complexity table.

Similarly, "A Comparative Study between Various Sorting Algorithms " by Jehad Hammad provided inspiration for creating visualizations of the algorithms' behavior across various scenarios.

6 Conclusions and Future Work

In this study, we addressed the problem of evaluating the performance of sorting algorithms under various conditions, including random, almost sorted, and reverse-sorted elements. The paper presents a comprehensive analysis of the execution times of seven sorting algorithms across different dataset sizes. The aim of this study was to provide insights into the efficiency of these algorithms and their suitability for different types of data.

We successfully demonstrated that certain algorithms, such as Radix Sort, Quick Sort, and Heap Sort, offer competitive performance across all datasets.

One major difficulty was structuring the code to handle multiple datasets at once. This challenge was overcome by redoing the code architecture. Unfortunately, I wasn't able to implement a more precise timing mechanism on my computer, which could have accurately measured even the smallest differences in execution time for datasets smaller than 1000 elements.

While we've covered sorting algorithms based on their execution time, future work could involve analyzing their space complexity as well. Additionally, including more sorting algorithms would provide a more complete picture of their performance.

References

[1] Course 2023, Semester I, Prof. D. Onchis. *"Algorithms and Data Structures I"*.

[2] Khalid Suleiman Al-Kharabsheh, Ibrahim Mahmoud AlTurani, Abdallah Mahmoud Ibrahim AlTurani, and Nabeel Imhammed Zanoon, *"Review on Sorting Algorithms, A Comparative Study"*, Available online: **Link**.

[3] Naeem Akhter, Muhammad Idrees, and Furqan-ur-Rehman, *"Sorting Algorithms – A Comparative Study"*, Available online: **Link**.

[4] Jehad Hammad, *"A Comparative Study between Various Sorting Algorithms"*, Available online: **Link**.

[5] The source code for the algorithms used can be found on GitHub, @Cata039, Available online: **Link**.