A theoretical and experimental comparison of sorting algorithms

Oriana Maria Iancu
Department of Computer Science,
West University of Timișoara, România
Email: oriana.iancu04@e-uvt.ro

May 12, 2024

Abstract

This paper presents a theoretical and experimental analysis of seven sorting algorithms—Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Tim Sort, and Heap Sort—across arrays of varying sizes and types. The study assesses each algorithm's execution time using lists that are reversed in order, almost sorted, and randomly sorted. The results offer significant insights for algorithm selection and optimization in computational tasks and further our understanding of the behavior of sorting algorithms.

Contents

List of Figures

1 Introduction

In the field of computer science, efficiency is the primary goal of algorithm design and optimization. Sorting algorithms, in particular, are fundamental components in this pursuit, with the essential objective of arranging data in a specific order.

From the early days of computing to the current era of big data and complex analytics, sorting algorithms have been essential tools in a wide range of applications, from information retrieval to computational biology. The demand for effective sorting techniques has only increased as computing power has grown, and datasets have risen a lot in size and complexity.

1.1 Motivation

The problem of comparing and evaluating sorting algorithms is important because it highlights the necessity of making informed decisions based on the requirements and constraints of certain projects, balancing factors such as time complexity, space complexity, implementation complexity, and practical constraints.

In this paper we are considering the following sorting methods:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Tim Sort
- Heap Sort

1.2 Informal description of solution

The solution to this problem relies on the implementation, as discussed later. Our experiment will explore the variation in execution time among different sorting methods under varying scenarios, determined by the size of the array.

1.3 Informal example

The variety of approaches and techniques covered by these seven sorting algorithms is fairly wide. The elementary algorithms: Insertion Sort Selection Sort and Bubble Sort are simple algorithms that are suitable for small datasets but show poor performance for larger inputs. Quick Sort and Merge Sort are based on the Divide and Conquer strategy and offer improved efficiency for larger datasets. Tim Sort was derived from Merge Sort and Insertion Sort and was designed to perform well on many kinds of data. Heap Sort is similar to the Selection Sort, but it is based on Binary Heap data structure.

1.4 Declaration of originality

Although the information in this paper is derived from a variety of relevant sources, the comparisons and analyses that follow are the product of the author's research and observations. To create a thorough comparison of sorting algorithms, the author has carefully synthesized relevant information from these sources to construct an extensive comparison of sorting algorithms. Furthermore, the paper contains personal observations, informed by the author's knowledge of this subject. Note that every reference used in this work has been properly acknowledged and credited.

1.5 Reading instructions

This paper delves into the evaluation of sorting algorithms to determine their suitability for various scenarios. First, we formally define the problem: choosing the best sorting strategy considering factors like dataset characteristics and execution time. Following that, we suggest a solution that involves a comparison of seven sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Tim Sort, and Heap Sort. An examination of their time complexity sets the stage for our experimentation, which involves sorting datasets of different sizes and types. We delve into the implementation details, including the organization of sorting algorithms in C programming language and the generation of diverse datasets. Then, we investigate these algorithms' performance on random, nearly-sorted, and reversed-order lists through rigorous experimentation. Additionally, we compare our findings with related works, highlighting both similarities and distinctions. We conclude by summarizing our findings, talking about possible future study directions, and outlining possible research topics.

2 Formal Description of Problem and Solution

The problem we're facing is figuring out which sorting method to use for different situations. For instance, certain algorithms may be really fast yet use a lot of memory, whereas other algorithms may be slower but more memory-efficient. Furthermore, while some methods might maintain the order of equal items, others might not. Finding the most effective sorting method thus comes down to evaluating time and space complexity, stability, and the size and nature of the data.

The solution we're proposing is a comparison of sorting methods across the execution time and the inherent characteristics of the data. Then, based on the experiment, we will be able to determine which sorting technique is best to use in different scenarios.

Now let's have a look over the time complexity of the seven sorting algorithms we'll conduct our experiment on:

Algorithms	Best case	Worst case	Average case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Tim Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Although $O(n \log n)$ denotes a highly efficient algorithm, $O(n^2)$ represents a significantly less favorable runtime. By looking at the average case we can observe that the 3 iterative sorting methods: Bubble Sort, Selection Sort, and Insertion Sort are not that efficient for large datasets.

So, simple sorting algorithms are very efficient when sorting a small amount of data. However, as the size of the array increases, the efficiency decreases. If we look at the 'Divide and Conquer' algorithms: Quick Sort, Merge Sort, and Tim Sort; we can see that they have the complexity of $O(n \log n)$ in almost all of the cases, showing that they are more appropriate when working with larger datasets.

3 Model and Implementation of Problem and Solution

The core of our solution lies in the implementation of seven sorting algorithms in the C programming language. The algorithms can be found at this GitHub link (**link here**). Each algorithm is meticulously crafted to ensure efficiency and accuracy in sorting various types and sizes of data.

In addition to the sorting algorithms, specialized algorithms for generating lists of varying sizes and characteristics have been developed.

To conduct this test effectively, the user must engage with each sorting algorithm individually, as each is implemented in its own distinct C file. The user must modify these files to change the output file according to the guidelines that are included. After that, the user can choose from the several list-generation algorithms that are available, each implemented in a different C file. These generators enable the creation of diverse datasets crucial for evaluating sorting algorithms. In addition to modifying the output file and selecting a list generator algorithm, the user is also required to adjust the size of the array generated, as specified in the comments within the C files. Performing this step guarantees testing flexibility and enables sorting algorithms to be evaluated on different sizes of datasets.

Once the list is generated, the user can proceed to execute one of the sorting algorithms of choice. It's imperative to note that each sorting algorithm is encapsulated within its own C file, ensuring modularity and ease of management. Finally, the user must record and analyze the execution time to draw meaningful conclusions regarding the efficiency and performance of the sorting algorithms under consideration.

4 Case Studies/Experiment

In this section, we examine the sorting algorithms' performance on datasets of various sizes (1000, 50000, 100000, 500000 elements) and types (random, nearly sorted, reverse-ordered arrays). Our goal is to gain a better understanding of algorithmic behavior under various scenarios through thorough experimentation.

4.1 Sorting a random-generated list

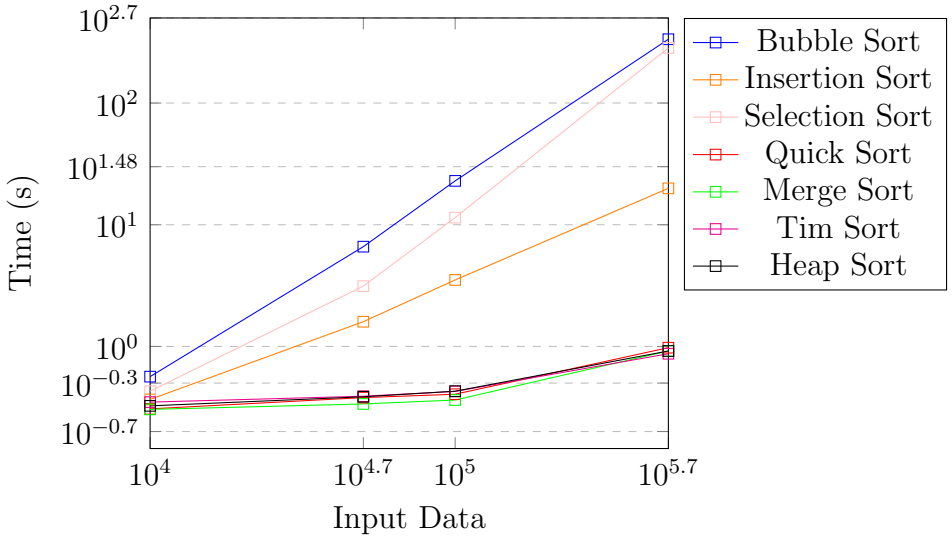

Figure 1: Size of the array and execution time (random generated list)

Input Data Size	10000	50000	100000	500000
Bubble Sort	0.565s	6.605s	22.929s	334.242s
Selection Sort	0.429s	3.13s	11.445s	283.482s
Insertion Sort	0.367s	1.597s	3.519s	19.914s
Quick Sort	0.307s	0.38s	0.405s	0.977s
Merge Sort	0.304s	0.337s	0.363s	0.918s
Tim Sort	0.349s	0.39s	0.427s	0.871s
Heap Sort	0.325s	0.386s	0.429s	0.918s

Figure 2: Execution time for sorting random-generated lists

We're starting our experiment with the sorting of some randomly generated lists. The following assumptions are based on the values we obtained in the table and graph above:

Significant differences in execution times can be seen between the Divide and Conquer methods, which include Quick Sort, Merge Sort, and Tim Sort, and the iterative sorting techniques, which include Bubble Sort, Insertion Sort, and Selection Sort. This is expected based on our earlier analysis of the time complexity of these algorithms. Compared to the Divide and Conquer procedures, which work more efficiently, the iterative

methods show noticeably slower performance, especially as dataset size increases. Also, notice that Insertion Sort was the fastest among the iterative sorting methods.

Furthermore, Heap Sort is highlighted by its remarkable efficiency, even when dealing with enormous arrays.

Concluding this case scenario, it's reasonable to assume that multiple sorting algorithms could efficiently handle the task on a small array of randomly sorted elements. However, in this scenario, Tim Sort excels when sorting larger arrays, while Merge Sort proves most efficient for smaller ones.

4.2 Sorting an almost sorted list

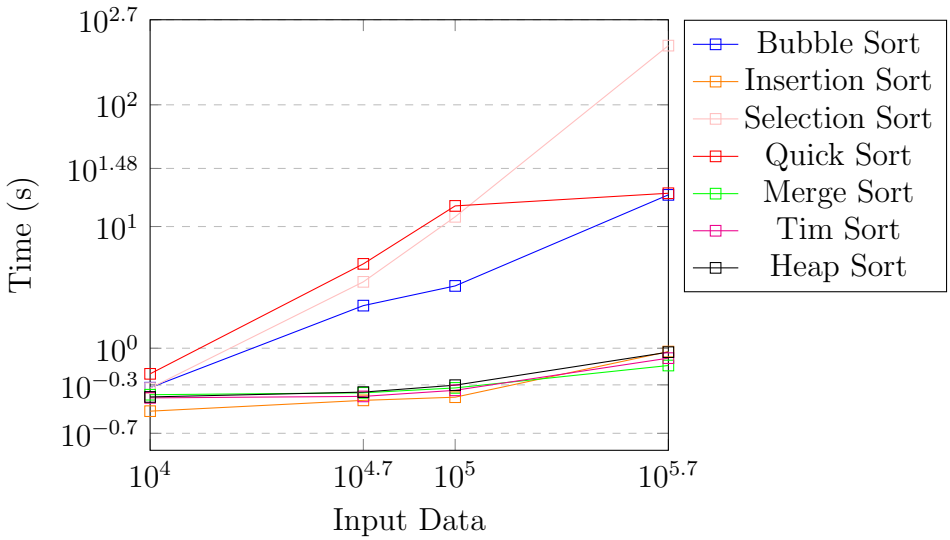

Figure 3: Size of the array and execution time(almost-sorted list)

Input Data Size	10000	50000	100000	500000
Bubble Sort	0.474s	2.239s	3.25s	18.204s
Selection Sort	0.47s	3.511s	12.016s	306.454s
Insertion Sort	0.304s	0.373s	0.396s	0.939s
Quick Sort	0.616s	4.932s	14.769s	18.769s
Merge Sort	0.415s	0.43s	0.472s	0.721s
Tim Sort	0.391s	0.402s	0.451s	0.829s
Heap Sort	0.398s	0.436s	0.497s	0.928s

Figure 4: Execution time for sorting almost-sorted lists

The differences become even more noticeable when almost-sorted arrays are sorted. When it comes to Insertion Sort, Merge Sort, and Tim Sort their speeds are very close to each other. However, Insertion Sort performs better than all the other sorting methods when sorting up to one hundred thousand elements. It also performs more efficiently when compared to its performance on randomly created lists. It takes advantage of the fact that it builds the final sorted list one element at a time. Each element of the array, beginning with the second (at index 1) is compared to the elements to its left and then inserted into the sorted sub-list so that it remains ordered. Even Bubble Sort shows improvement under these conditions, but Selection Sort continues to exhibit suboptimal performance.

Unexpectedly, Quick Sort performs worse than Bubble Sort in terms of execution time, declining dramatically. This departure from its usual efficiency highlights how sensitive Quick Sort is to the array's starting condition.

Concluding, Insertion Sort would be the most efficient sorting algorithm when sorting smaller arrays. However, Merge Sort is the fastest for arrays of five hundred thousand elements.

4.3 Sorting a reverse ordered list

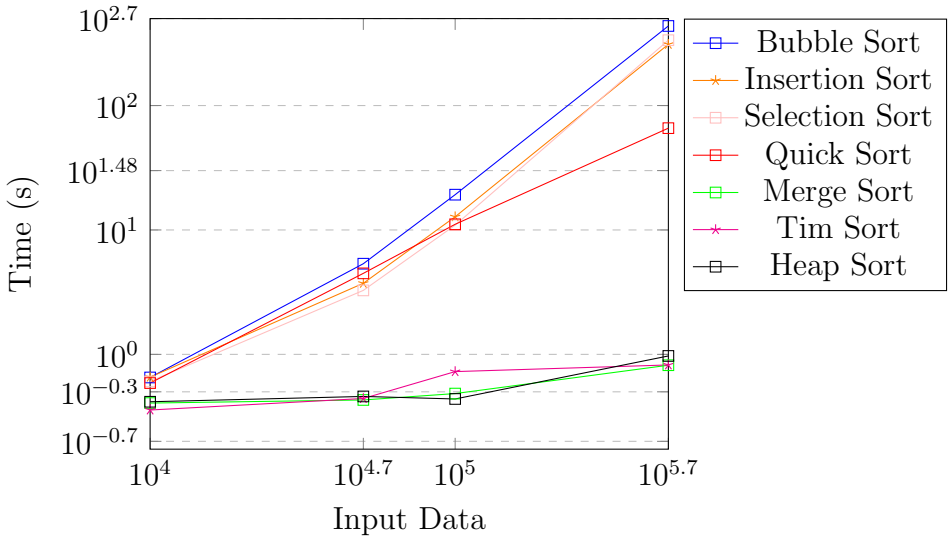

Figure 5: Size of the array and execution time (reverse ordered list)

Input Data Size	10000	50000	100000	500000
Bubble Sort	$0.652s$	$5.354s$	$19.213s$	$436.158s$
Selection Sort	$0.606s$	$3.268s$	$10.929s$	$335.285s$
Insertion Sort	$0.656s$	$3.728s$	$12.766s$	$310.562s$
Quick Sort	$0.588s$	$4.479s$	$11.107s$	$65.809s$
Merge Sort	$0.406s$	$0.43s$	$0.484s$	$0.818s$
Tim Sort	$0.357s$	$0.442s$	$0.727s$	$0.822s$
Heap Sort	$0.416s$	$0.459s$	$0.478s$	$0.971s$

Figure 6: Execution time for sorting reverse ordered lists

As we can observe in the figures above, sorting a reversed-ordered array of five hundred thousand elements brings a dramatic shift in performance for Bubble Sort, Selection Sort, Insertion Sort, and even Quick Sort.

Although Merge Sort, Tim Sort, and Heap Sort maintain their efficiency for larger arrays, we can observe an increase in time when sorting arrays of ten thousand, fifty thousand, and one hundred thousand elements. Additionally, the execution time of Tim Sort increases dramatically when sorting one hundred thousand elements compared to the cases analyzed previously.

In this scenario, the disparity between Insertion Sort and the faster algorithms like Tim Sort, Heap Sort, and Merge Sort becomes quite evident. Notably, Tim Sort, despite being a hybrid algorithm based on Merge Sort and Insertion Sort, doesn't quite match the swiftness of Merge Sort itself.

This time, Merge Sort proves to be the most suitable choice when sorting a reversed-order array of any size.

5 Related Work

The main goal of our research is to investigate the way alternative sorting algorithms perform on various volumes and types of datasets. We discover notable variations in the execution times of iterative sorting algorithms and Divide and Conquer methods, supporting well-established notions of algorithmic efficiency. Furthermore, our findings about the behavior of Insertion Sort and Merge Sort on almost-sorted arrays provide important information about algorithmic behavior in particular scenarios.

To gain a deeper understanding of the insights presented in our paper, we can refer to existing research on the comparison of sorting algorithms. By looking at relevant research in this area, we may draw attention to the advantages and disadvantages of our paper as well as the cited studies, providing a thorough analysis of the problem. In comparing our findings to existing research, we observe both similarities and distinctions.

Comparing to "A Comparative Study of Sorting Algorithms" written by D.R. Aremu, O.O. Adesina, O.E. Makinde, O. Ajibola, and O.O. Agbo-Ajala (**link here**), our analysis focuses only on the execution time of each algorithm. Their paper gives a thorough analysis of sorting algorithms, exploring their working processes, time complexity, and space complexity; in contrast, our study focuses on the differences in execution time when sorting different datasets. Moreover, our research includes the analysis of seven different sorting algorithms, whereas their analysis only looks at three. Even though our study is more limited, the more algorithms we include, the more we may learn about how well they perform in comparison.

A benefit of our methodology is the extensive scope of our testing, encompassing a broad spectrum of dataset sizes and types. However, a potential limitation lies in the specificity of our experimental conditions, which may not fully capture the complexities of real-world datasets. Additionally, further investigation is required to examine other variables that can affect the performance of sorting algorithms, even if our findings provide insightful information on algorithmic behavior.

Another example is "Performance Analysis of Sorting Algorithms" by Matej Hulin (**link here**). In addition to detailing the asymptotic time complexity of each algorithm, this paper explores various testing criteria and considers factors such as space complexity and stability. In real-world applications, addressing these factors is essential because they provide a thorough understanding of algorithm performance that goes beyond time complexity analysis.

6 Conclusions and Future Work

Our approach to analyzing execution times produced valuable insights into algorithmic performance. We have successfully tested and implemented seven sorting algorithms, providing a thorough performance comparison. By considering multiple dataset types and sizes, we provided a nuanced understanding of algorithmic behavior under varied conditions.

The investigation of execution time was the main emphasis of our work. However, it is still necessary to investigate other aspects of sorting algorithm analysis, such as stability and space complexity. Further research is warranted to investigate the performance of sorting algorithms under more diverse and real-world conditions.

Future research efforts could focus on incorporating space complexity and stability analysis into sorting algorithm evaluations.

7 References

Rizvi, Dr Qaim Rai, Harsh Jaiswal, Ragini. (2024). Sorting Algorithms in Focus: A Critical Examination of Sorting Algorithm Performance. 5-6

Yash Chauhan, Anuj Duggal(2020). Different Sorting Algorithms comparison based upon the Time Complexity. IJRAR September 2020, Volume 7, Issue 3, 120-121, 5-6

Adesina, Opeyemi. (2013). A Comparative Study of Sorting Algorithms. African Journal of Computing ICT. 6. 199-206, 4-6