A Theoretical and Experimental Comparison of Sorting Algorithms

Crina Bărbieru,
Department of Computer Science,
West University,
Timișoara, România
email: crina.barbieru04@e-uvt.ro

March 2023

Abstract

The problem of sorting arises frequently, since it can reduce the complexity of an algorithm. Nowadays, with the amount of data steadily increasing, it is important to recognize the advantages and disadvantages of certain sorting algorithms, in order to maximize time and memory efficiency.
The goal of this paper is to analyze and compare different sorting algorithms, both theoretically and experimentally, highlighting the contexts in which they are most useful.

Contents

1 Introduction

1.1 Motivation

Sorting is one of the fundamental operations in Computer Science, and at the same time one of the most well-studied problems. It can generally be understood as the operation of rearranging the objects in a given set in a particular order, or after a particular criterion.
Because sorting is a fundamental operation used in many instances, we can observe a great diversity of methods. There is no ideal sorting algorithm for all given instances.
Performance is influenced by many factors, so the purpose of our paper is to analyse different situations in which sorting might be needed, and decide which algorithm is optimal.

2 The Sorting Algorithms used in Experiments

In this section, we will present each of the chosen sorting algorithms, by describing its method, time and memory complexity. The algorithms will be grouped based on two criteria: complexity and type (comparison based vs. non-comparison based). Complexity will be evaluated using *The O-notation*, as introduced by Donald Knuth in [1].
The purpose of this section is to describe the algorithms, but also to set expectations and anticipate the results of the experiment. Even if some of these methods are familiar to the readers, it is important to compare them in order to gain a global view on the issue of sorting.

2.1 Comparison Based Algorithms

As suggested, these algorithms rely on comparisons between the values of elements in order to sort a given array. Although it seems an intuitive approach, it has its limitations, especially when it comes to time complexity. We will highlight these limitations further in the paper.

2.1.1 Selection Sort

Selection Sort is a comparison-based algorithm which is typically implemented to repeatedly find the minimum element in the array, which is split into two subarrays. At each iteration, the minimum element in the unsorted subarray is computed and placed in its appropriate spot, therefore "extending" the sorted subarray.
In terms of complexity, Selection Sort is $O(n^2)$ for the best, worst and average cases. In terms of memory, this method does not need any additional memory space for the elements in the array. All the modifications take place in the array. While there have been improvements in multiple implementations

of Selection Sort in order to reduce its time complexity, we will analyze the classical algorithm in this paper, which is additionally explained in [2].

2.1.2 Insertion Sort

Similarly to Selection Sort, this technique works by dividing the array into two subarrays, but unlike the former, it does not compute the minimum at each step. Instead, it places each element into its correct position.
The worst and average case have a complexity of $O(n^2)$, while the best case runs in linear time, as highlighted in [2]. All of the changes are completed into the array, therefore no additional memory is needed for the array. Moreover, it is a stable algorithm, because it preserves the relative order of equal values.

2.1.3 Bubble Sort

Bubble sort is one of the simpler and most popular sorting algorithms. It works by swapping consecutive elements if they are out of order. This operation is repeated until no more swaps need to be performed. The number of swaps done during Bubble Sort is equal to the number of inversion in the given list, as described in [2].
The algorithm has a worst-case complexity of $O(n^2)$. No additional memory is required besides the variables used in the for loops, since all swaps are performed in the list. It is a stable algorithm, if two elements are equal they are not swapped, therefore the relative order is preserved.

2.1.4 Merge Sort

Merge Sort [2] is a divide-and-conquer algorithm. First, the unsorted array is divided recursively into multiple subarrays, each containing one element. Because a one-element array can be considered sorted, those subarrays are repeatedly merged. Therefore, through dividing and merging, we obtain the sorted array.
The merge operation between two lists is done in $O(n)$ time. The best, worst and average case complexity of the entire algorithm is $O(n * log(n))$. From the memory point of view, it needs an $O(n)$ amount of space to copy the elements while sorting them. It is a stable sorting algorithm.

2.1.5 Quick Sort

Quick Sort is also a divide-and-conquer algorithm, but it has a different approach compared Merge Sort. From the given list, an element is chosen, which is also called a pivot. Then a partition is being done: elements are arranged into two subarrays, one contains the elements smaller than the pivot, and the other contains bigger elements. This operation is applied recursively to the obtained subarrays. This method was invented by Charles Antony Richard Hoare and explained in [3].

On average, the algorithm has a $O(n * log(n))$ time complexity. In the worst case scenario, where we obtain very unequal splits, Quick Sort runs in $O(n^2)$ time. With most implementations, Quick Sort is not stable.

2.1.6 Heap Sort

This technique uses the heap data structure, which is an array object that we can view as a nearly complete binary tree [2]. The implementation uses a max-heap. After the heap is built, the largest elements from the heap is repeatedly removed and inserted into a sorted array. The heap is updated after each step, in order to maintain its max-heap property. The sorted array is obtained after all elements have been removed from the heap.
The worst-case scenario has a complexity of $O(n * log(n))$. It is not a stable sorting algorithm.

2.2 Non-Comparison Based Algorithms

We must question: "Is $O(n * log(n))$ the best we can do when it comes to sorting?". For comparison-based algorithms, the answer is yes. We can view a comparison technique as a decision tree, where the length of the longest path from the root to a leaf represents the worst-case number of comparisons. As explained in [2], we obtain an $\Omega(n * log(n))$ lower bound for the complexity of comparison-based sorting algorithms.
On the other hand, other algorithms can be designed that do not require comparison between elements and have better worst-case complexities. Some examples of such algorithms will be discussed in this section.

2.2.1 Counting Sort

Counting sort [2] is an algorithm which sorts an array by counting (as the name suggests). Each value's place in the array is computed, by knowing how many other values have to come before it, using an auxiliary array.
Because the algorithm only uses simple for loops for the initialization of the auxiliary array, computing the partial sums and arranging the elements into the correct order, the worst-case complexity is $O(n + k)$, where k represents the range of values in the given array. We notice the trade-off between time and memory.

2.2.2 Radix Sort

This sorting method sorts the elements based on their digits, starting from the least significant digit. The process is repeated for each digit, while preserving the relative order from the previous steps for numbers that have, at some point, equal digits.
The worst, average and best case complexity of Radix Sort is $O(d * (n + k))$, where d represents the number of digits of each value, n is the size of the array and k is the number of values each digit can take [2].

2.2.3 Bucket Sort

Bucket Sort [2] is an algorithm which places the elements from the array in different buckets. Each of the buckets is sorted, then the buckets are arranged in the correct order and the elements are put back in the array. It is somewhat similar to Radix Sort, in which case the buckets would represent digits at each step.
Although the worst-case scenario has complexity $O(n^2)$, the average case runs in linear time. When it comes to memory, it needs additional space for creating the buckets and sorting them, therefore having a space complexity of $O(n + k)$, where k is the number of used buckets.

3 Implementation of the Algorithms

3.1 Methods

In order to make the experiments possible, we implement all the algorithms described above as functions, in Python. We will read the data sets were from a text file, and the output consisting of the sorted array and two time-measurements will also be stored in a file.
The source code and test cases can be found here:
`https://github.com/crinabarbieru/Sorting_Algorithms_Analysis.git`

3.2 Time measurement

The purpose of the experiment is to measure the time taken by the sorting algorithms on different inputs. At first, I imported two functions from the Python time library, called *time*() and *process_time*(). The difference between the time functions is: *process_time*() returns the time taken by the system and the user CPU, while *time*() takes into consideration other running programs as well. While doing the experiment, I noticed the differences between these two functions were not significant - it was only about one second for the biggest data set. Therefore, in the tables which will describe experiment results, we will only consider the results of the *time*() function. Probably on larger data sets and in other real life situation, there would be a need to present both of these functions.
The time function is called twice, before and after the chosen sorting function. The run time of the algorithm will be determined by the difference between the values of the two calls.

3.3 Architecture

The experiments were conducted on a machine with Intel(R) Core(TM) i3-5005U CPU 2.00 GHz, 8GB RAM, Windows 10 Pro, and the program used for the Python code is Pycharm 2022.3.3

4 Test Cases

We will test out algorithms on four types of inputs: random arrays, sorted in ascending order, sorted in descending order, and partially sorted arrays. The arrays contain positive integers, ranging from 1 to 10^6. The data sets have between 10^2 to 10^5 elements. For the partially sorted input, about 5% - 10% of elements are not in their correct order.
We choose these test cases in order to highlight the best, worst and average scenarios for our chosen sorting algorithms.

5 Analysis of Results

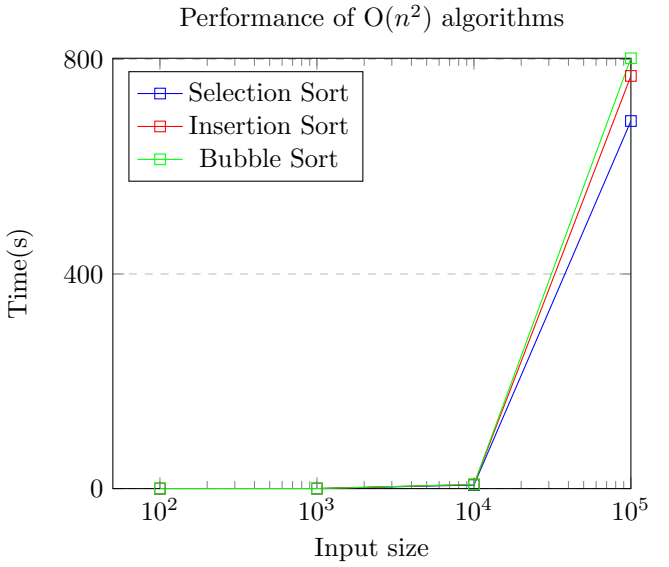
We can observe the results in the following tables and charts, separated according to the different test cases. In tables, we group algorithms based on their time complexity.

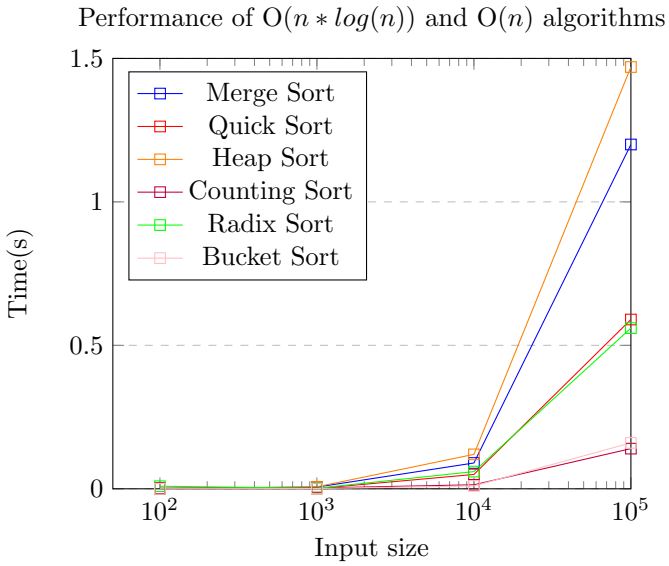
5.1 Case 1: Random Arrays

		time(s)		
List	Elements	Selection Sort	Insertion Sort	Bubble Sort
L11	100	0.00100	0.00120	0.00100
L12	1000	0.06959	0.07756	0.07129
L13	10000	6.6597	7.71783	7.77843
L14	100000	685.314	769.325	802.469

		time(s)		
List	Elements	Merge Sort	Quick Sort	Heap Sort
L11	100	0.00100	0.00099	0.00099
L12	1000	0.00616	0.00137	0.00768
L13	10000	0.09639	0.04941	0.1228
L14	100000	1.2103	0.59574	1.47876

		time(s)		
List	Elements	Counting Sort	Radix Sort	Bucket Sort
L11	100	0.00276	0.00990	0.0000001
L12	1000	0.00342	0.00282	0.00178
L13	10000	0.01414	0.061811	0.01320
L14	100000	0.14438	0.56094	0.15975





In this case, most algorithms performed as expected.We can see that for small inputs, the running times of algorithms are somewhat comparable, regardless of their complexity. Algorithms that have a complexity of O(n^2) run faster on the first data set than Radix Sort and Bucket Sort. The fastest algorithm for the first data set was Bucket Sort.

As the number of elements starts growing, we can see the differences between algorithms. Those that have complexity of O(n^2) need a lot more time to sort inputs of 10^4 and 10^5 elements, taking up to 800 seconds. List L14 really portrays the importance of time complexity, since algorithms in the last category

are able to sort it more than 1000 times faster.

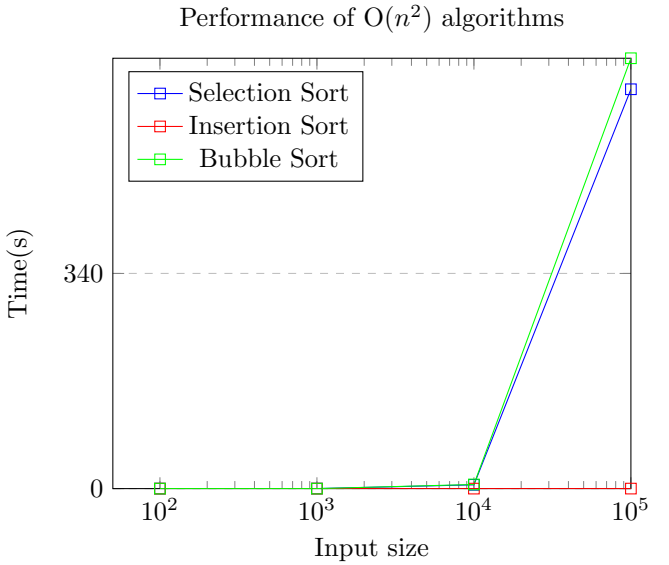
5.2 Case 2: Sorted Arrays in Ascending Order

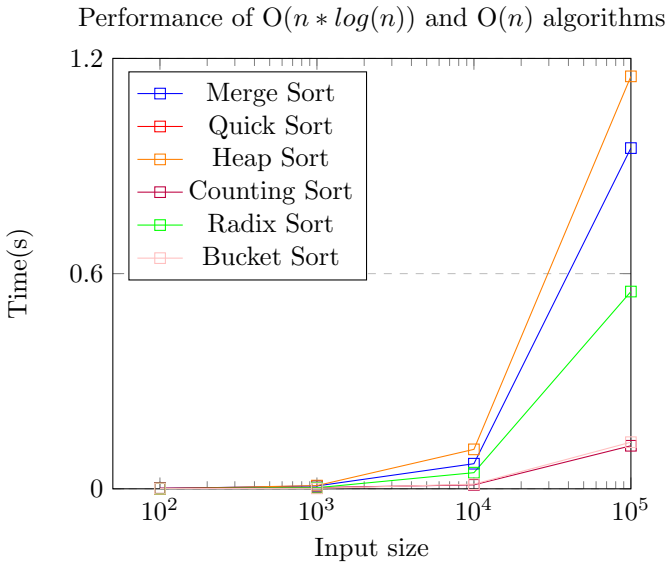
		time(s)		
List	Elements	Selection Sort	Insertion Sort	Bubble Sort
L21	100	0.00099	0.000001	0.00006
L22	1000	0.06014	0.00003	0.006443
L23	10000	6.00924	0.003066	6.52142
L24	100000	631.5633	0.035241	680.6475

		time(s)		
List	Elements	Merge Sort	Quick Sort	Heap Sort
L21	100	0.00100	0.001604	0.00081
L22	1000	0.00817	RecursionError	0.009884
L23	10000	0.07382	RecursionError	0.116461
L24	100000	0.95476	RecursionError	1.155673

		time(s)		
List	Elements	Counting Sort	Radix Sort	Bucket Sort
L21	100	0.002863	0.000788	0.0006
L22	1000	0.003612	0.003038	0.001216
L23	10000	0.011248	0.045063	0.012886
L24	100000	0.117748	0.5559	0.127287

In this case, most algorithms performed as expected.One exception is Quick Sort, where an error occurred. Starting from the second data set, Quick Sort could not sort the lists because of a "RecursionError: maximum recursion depth exceeded in comparison". This portrays not only the limitations of Python, but also the inefficiency of Quick Sort when it comes to already sorted inputs. By choosing the last element as a pivot, we obtain an unfavorable partition each time.
Surprisingly, the fastest algorithm in this case was Insertion Sort, which was even faster that non-comparison based algorithms. There is a reason behind this: already sorted inputs constitute Insertion Sort's best case scenario, when the algorithm runs in linear time. Because each element is larger than the previous one, no swaps are done and the array is only being moved through once.



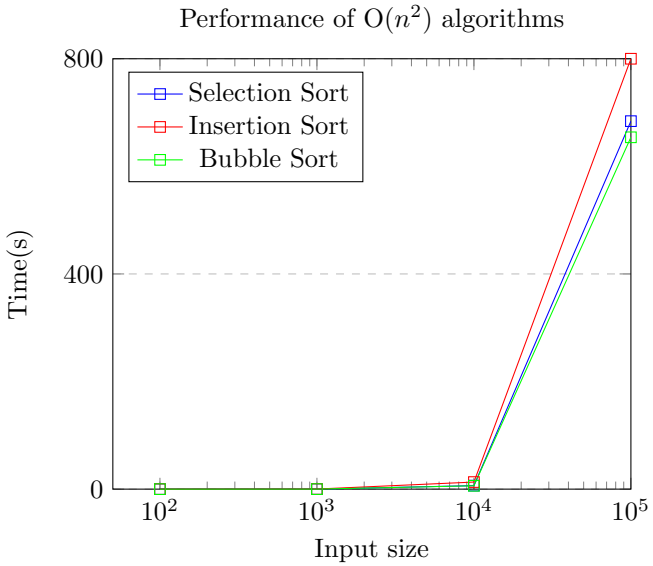


5.3 Case 3: Sorted Arrays in Descending Order

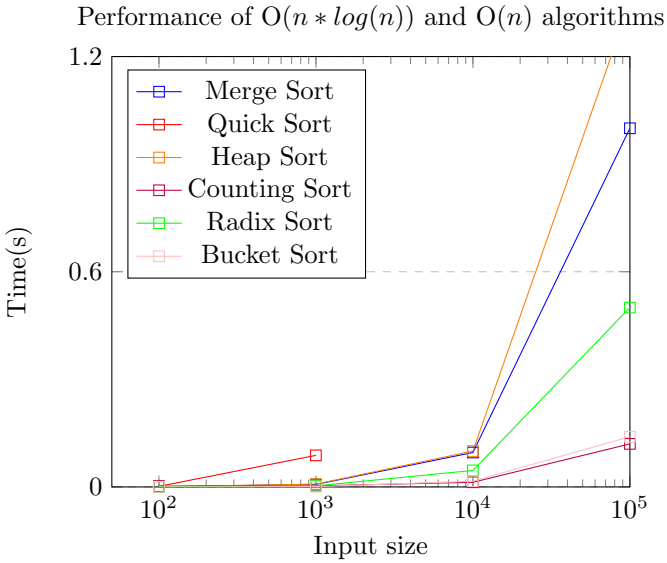
		time(s)		
List	Elements	Selection Sort	Insertion Sort	Bubble Sort
L31	100	0.001256	0.000816	0.0009
L32	1000	0.065342	0.17246	0.05605
L33	10000	6.54473	13.15625	6.22166
L34	100000	684.6569	800	654.2947

time(s)				
List	Elements	Merge Sort	Quick Sort	Heap Sort
L31	100	0.000826	0.001604	0.00053
L32	1000	0.007442	0.088687	0.008447
L33	10000	0.096275	RecursionError	0.107625
L34	100000	1.02951	RecursionError	1.354182

time(s)				
List	Elements	Counting Sort	Radix Sort	Bucket Sort
L31	100	0.002387	0.000389	0.0001
L32	1000	0.003097	0.003784	0.00112
L33	10000	0.013124	0.046851	0.016145
L34	100000	0.11997	0.49872	0.143125



In this case, the algorithms performed as expected. On smaller inputs, the O(n^2) algorithms had comparable running times with the ones of O($n*log(n)$) complexity. The difference became noticeable starting with the third list L33, where they ran about 100 times slower.
The elements were no longer sorted in the correct, ascending order and Insertion Sort was the least efficient out of the O($n * log(n)$) algorithms. The Recursion Error occurred again for Quick Sort in the last two cases (L33 and L34). Since the pivot is the last element in the array, it is also the smallest one. Therefore, we obtain an inefficient partition, an unbalanced split of the array.
Counting Sort and Bucket sort were the most efficient algorithms, taking a little over a tenth of a second for the largest input. Radix Sort was about 4 times slower than Counting Sort.

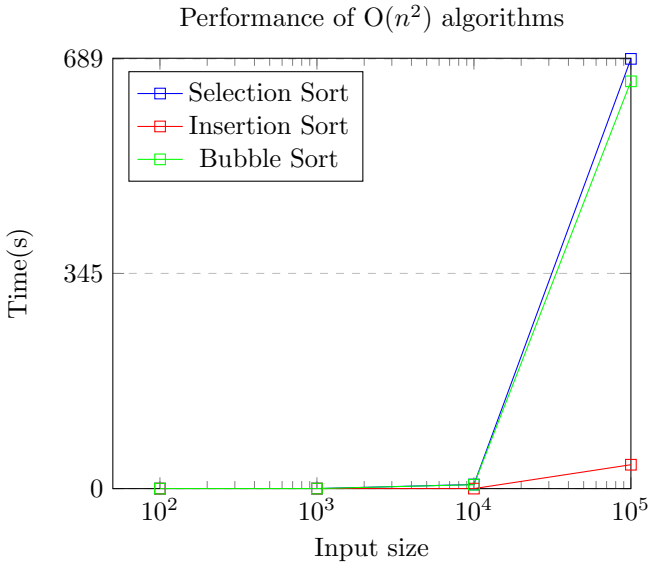


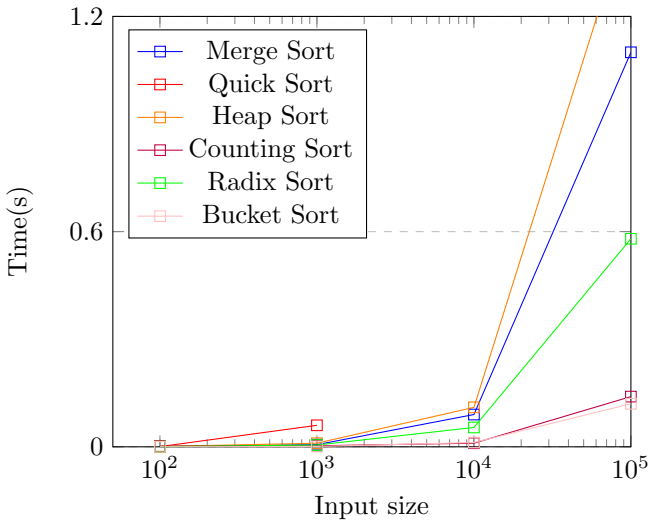
5.4 Case 4: Partially Sorted Arrays

time(s)				
List	Elements	Selection Sort	Insertion Sort	Bubble Sort
L41	100	0.00059	0.0001	0.000587
L42	1000	0.06687	0.014182	0.06885
L43	10000	6.62652	0.02624	6.31614
L44	100000	689.857	38.34872	653.9756

time(s)				
List	Elements	Merge Sort	Quick Sort	Heap Sort
L41	100	0.000587	0.000697	0.000988
L42	1000	0.006074	0.06079	0.01018
L43	10000	0.089327	RecursionError	0.11376
L44	100000	1.09985	RecursionError	1.50205

time(s)				
List	Elements	Counting Sort	Radix Sort	Bucket Sort
L41	100	0.00209	0.000558	0.000001
L42	1000	0.00311	0.00501	0.001023
L43	10000	0.0143	0.054716	0.012234
L44	100000	0.13703	0.581862	0.1235





In this case, algorithms performed better for the partially sorted arrays than they did in Case 1 or Case 3. As expected, they were still slower than in Case 2, when the arrays were already sorted.
Insertion Sort is the most efficient among O(n^2) algorithms, the difference becomes somewhat significant when the input size is larger than 10^3. Selection Sort and Bubble Sort perform similarly, but there is a noticeable difference only for list L41.
When it comes to O($n * log(n)$) algorithms, Quick Sort performed better than in the last two cases, but was still the slowest out of the three, yielding

a Recursion Error for larger data sets (L43 and L44). Merge Sort was slightly faster than Heap Sort in all instances.
Similarly to other cases, Counting Sort and Bucket Sort had similar running times, and were still about 5 times faster than Radix Sort.

6 Analysis of Related Work

There are multiple papers that have analysed sorting algorithms, many of them with similar findings.
In [4] multiple improved and classical sorting techniques are analysed, some of which are also present in our paper. Each technique's advantages and disadvantages are presented, in a detailed comparative analysis based on time and space complexity, stability, approach and applications. While my paper was centered on the experimental part, the findings are similar in terms of efficiency and performance.
Selection Sort and Insertion Sort are compared in [5] through experiment, using fully sorted, partially sorted and unsorted arrays with up to 10^5 elements. In the paper, Insertion Sort is a lot faster to Selection Sort in already sorted inputs, while the latter outperforms the former in the case unsorted inputs of all sizes. The findings in our paper are similar, with Insertion Sort being visibly faster than Selection Sort in Case 2 and Case 4 - sorted and partially sorted arrays.
In [6], the same sorting techniques we have also chosen for our paper are analysed. However, their implementation differs in some aspects, such as the chosen programming language or the size of the data sets. The paper also contains a more in-depth analysis, not only from the point of view of time complexity, but also with a focus on space complexity. The findings of [6] are similar to the ones in our paper, but additionally the author has implemented some improved sorting algorithms.

7 Conclusion

The purpose of this paper was to analyse different sorting techniques, both from a theoretical and an experimental point of view. The findings of this paper show that the efficiency of a sorting technique is influenced by multiple factors, such as time and space complexity, data sets and implementation. The experimental part was key in understanding the real life applications of the theoretical part.
The chosen case studies point to the fact that there is no universal, "one fits all" algorithm. Some of the easier techniques, such as Selection Sort or Bubble Sort, work well for smaller inputs, while they are clearly outperformed by the $O(n * log(n))$ and $O(n + k)$ algorithms as the input size gets larger and larger. There were some unexpected results at first sight, but eventually they could all be explained and supported by the theoretical knowledge.

As technology advances, the amount of data and information continues to grow steadily, and there will probably be a need for optimization and improvement of sorting algorithms. At the same time, simple methods should not be overlooked, since they are at the very basis of so many other complex algorithms.

References

[1] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[3] Charles AR Hoare. Quicksort. *The computer journal*, 5(1):10–16, 1962.

[4] Purvi Prajapati, Nikita Bhatt, and Nirav Bhatt. Performance comparison of different sorting algorithms. *vol. VI, no. Vi*, pages 39–41, 2017.

[5] Fahriye Gemci Furat. A comparative study of selection sort and insertion sort algorithms. *International Research Journal of Engineering and Technology (IRJET)*, 3(12):326–330, 2016.

[6] Ashok Kumar Karunanithi et al. A survey, discussion and comparison of sorting algorithms. *Department of Computing Science, Umea University*, 2014.