A Theoretical and Experimental Comparison of Sorting Algorithms

Marius-Alexandru Roșu
Department of Computer Science,
Faculty of Mathematics and Informatics,
West University of Timișoara, Romania
Email: ma.rosu06@gmail.com

June 2, 2023

Abstract

We live in a world that is constantly changing and improving. In our century, speed is now the key to everything around us, from navigating the internet to developing programs and algorithms.

The power of something lies in its time efficiency. Is it fast? Can it be faster? How fast can it be if we do this? These are the questions that characterize the effectiveness of something.

Contents

1 Introduction 3

2 Formal Description of Problem and Solution 4

3 Model and Implementation of Problem and Solution 5

4 Case Studies/Experiment 6

5 Related Work 11

6 Conclusions and Future Work 11

1 Introduction

In nowadays there are a lot of applications and products that are lacking time and space efficiency and even if for humans is not necessarily visible, for a computer is and can lead to bigger consequences which can become a problem later. For example, if we try to do a little calculation we will see that if we have to do the same thing for 1000 times and each one takes 0.004s, we will notice that the time is stacking and it easily went from 0.004s to 4 entirely seconds and this can be crucial in some cases. Also there are moments where poor memory management can make your product have unexpected crashes. There are a lot of applications and web pages that have sorting systems such as “From lower to highest” or “From highest to lower”, each one of these systems is based on a sorting algorithm which we will talk about later. The existing solutions may have a bad time and/or space efficiency and they can slow your product a lot, especially if you don’t know which one suits your needs the most. In this paper we will go over the following sorting algorithms:

- Bubble Sort [1]
- Merge Sort [2]
- Insertion Sort [3]
- Quick Sort [4]
- Heap Sort [5]
- Radix Sort [6]
- Selection Sort [7]
- Counting Sort [8]

Each one of these algorithms have it’s upsides and downsides and we will go later over each one and talk about their performance and their complexity on different datasets.

For now let’s assume you have a stack of papers enumerated from 1 to 100, but in a random order, and you have to sort them in an ascending order. Maybe you will start to compare the adjacent papers and swap their places if they are in the wrong order. This method is called “Bubble Sort” and it might not be the best approach. Instead you think about taking the first paper and rearrange the other papers, one by one, based on their correct positions compared with the data already sorted. This method is called "Insertion Sort" and it will clearly be more efficient than the other. This is a commonplace example but in other cases, insertion sort might not be the best option either.

This paper intends to help readers choose the perfect sorting algorithm for their work and to get the best time/space efficiency. In this paper, we will not present any algorithm, but we will try them on small and bigger lists. It is expected that the readers are already familiar with all of the sorting algorithms from above.

This paper is an original work and all sources have been properly acknowledged and cited.

2 Formal Description of Problem and Solution

In order to get the most of these paper you must know that a sorting algorithm has his own properties and you must understand what is the difference between a comparison-based sorting algorithm and a non-comparison-based sorting algorithm, what time complexity, space complexity, adaptability and stability means and how they can influence your choice when picking the sorting algorithm for your product. Although we may still use one algorithm for any problem it is not really recommended. There are plenty of choices and each one of them has it's own advantages and disadvantages depending on what you have to do. Some of them are time efficient but not space efficient, others may be easier to implement but with a low efficiency.

If we talk about time efficiency we look at the time complexity. Some algorithms, such as Merge sort, have an $\mathrm{O}(n \log n)$ average case time complexity, while others, like Bubble Sort have an $\mathrm{O}(n^2)$ time complexity.

Space efficiency is also given by the space complexity, which refers to the amount of memory needed for the algorithm to finish the sorting. For example, Merge Sort has a high space complexity, while algorithms like Insertion Sort has a low space complexity due to the fact that the sorting happens in place.

The stability of a program is given by the ability of the sorting algorithm to keep the order from the input of two elements that have the same value.

For example, Merge Sort keeps the order of the equal elements, while Quick Sort doesn't. Stability is very useful in sorting databases where you need to record the order of the entries.

An algorithm is adaptive when it is capable of taking advantage of already sorted portions from the list. An algorithm, such as Insertion Sort, which is adaptive can have a significant performance improvements in practice.

Comparison-based sorting algorithms, such as Quick Sort, compares the value of the elements directly and rearrange them in the ascending order, while non-comparison-based sorting algorithms does not rely on comparing the value of the elements, but on the properties or characteristics of the elements.

Depending on your scenario some sorting algorithms may perform better than the others. For example, Merge Sort performs better than Insertion Sort on a large lists, while Insertion does a better job than Merge Sort in small lists.

In the experimental part we will go over each algorithm and we will put them to a test, we will work with each one on small lists and bigger lists to see how they behave and which one is the best option for that case.

3 Model and Implementation of Problem and Solution

For this chapter we will implement an example of a problem and we will resolve it with the Insertion Sort algorithm to understand better the entire process of sorting algorithms. It will be provided some simple steps so you can maybe try it yourself.

To start, we will choose a random array given in a completely random order: [4, 0, 3, 1, 2]. Our main goal is to rearrange this to be [0, 1, 2, 3, 4] and we will achieve that by using Insertion Sort.

In our computer implementation we need to have an input and output method, which in this case will be a text file where we will put the unsorted array from above, and the output should be shown in the console. Besides these we will also need to have a module for the sorting algorithm.

The input data will be stored in an array of 5 elements, and by the end of the program the same array will be modified into a sorted one. The Insertion Sort algorithm works by iterating over the array and inserting each element into it's correct position in the sorted sub-array to the left like following:

1. $[4, 0, 3, 1, 2]$
2. $[0, 3, 4, 1, 2]$
3. $[0, 1, 3, 4, 2]$
4. $[0, 1, 2, 3, 4]$

This experiment can be done (not only) in Code Blocks, using C++. First you need to install the IDE, write the sorting algorithm, in this case was Insertion Sort but you can choose another if you want, make a text file for input and then extract the data for the array from there. When the sort is completed the sorted array will be shown in the console and the output should look like this: [0, 1, 2, 3, 4].

4 Case Studies/Experiment

All of the sorting algorithms I presented in the first section of this paper are used in many cases in programming and we already know that each one of them has his own advantages and disadvantages. To see how they perform I ran each one of them on a computer in Code Blocks version 20.03, on a computer with i5-1135G7 @ 2.40GHz, 8 GB RAM, on 3 types of lists:

1. 5000 elements
2. 50000 elements
3. 500000 elements

For each algorithm and list we have 7 cases: a completely random array, an array with the first and last item inverted, with the first item in the last position, with the last item in the first position, with the first or last item somewhere in the middle of the array, and finally an already sorted array. It is important to note that the result you will see are an average between all of 7 cases and in your attempt to reproduce the experiment the result might differ a little.

Before analyzing the result of the experiment let's look at the time and space complexity of each algorithm in the table below.

Algorithm/Complexity	Bubble	Insertion	Selection
Time	n^2	n^2	n^2
Space	1	1	1
Stability	yes	yes	no
Adaptability	yes	yes	no
Comparison	yes	yes	yes
Recursivity	no	no	no

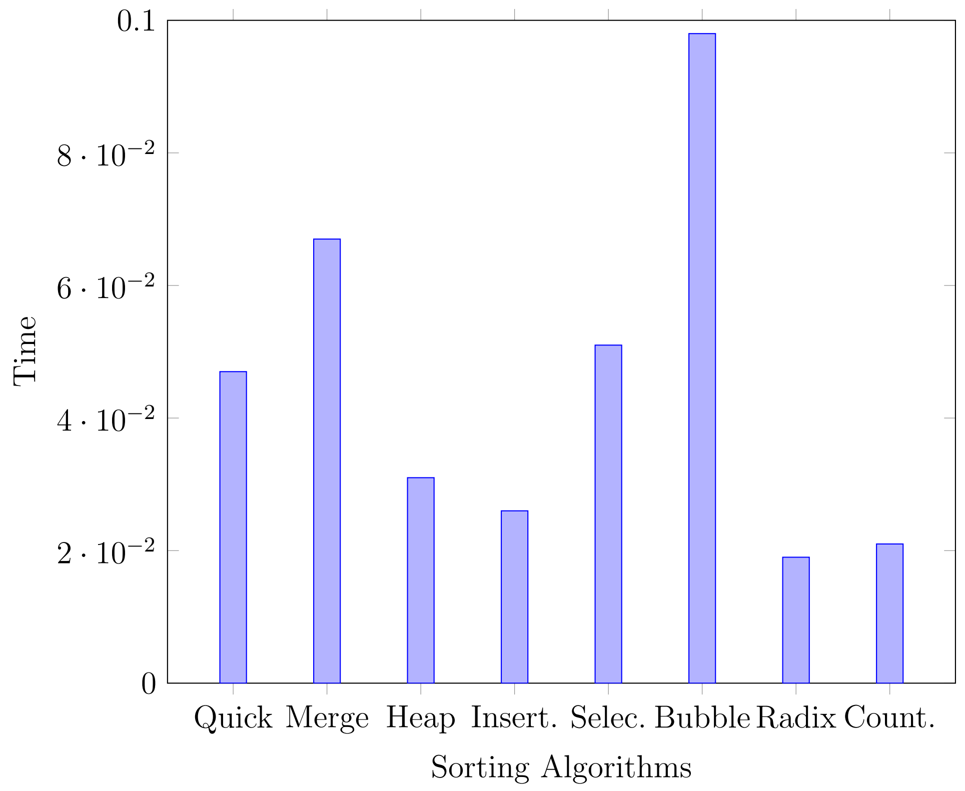
Algorithm/Complexity	Quick	Merge	Heap
Time	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
Space	$log_2 n$	n	1
Stability	no	yes	no
Adaptability	yes	no	no
Comparison	yes	yes	yes
Recursivity	yes	yes	no

Algorithm/Complexity	Counting	Radix
Time	$n + k$	$d(n + k)$
Space	k	n+k
Stability	yes	yes
Adaptability	yes	yes
Comparison	no	no
Recursivity	no	no

As we can see in the table above it is noticeable that non-comparison-based algorithms have in general a more efficient time complexity, while the exponential time complexity algorithms work better in terms of space complexity. It is important to know that in practical scenarios, the result may differ a little depending on each case.

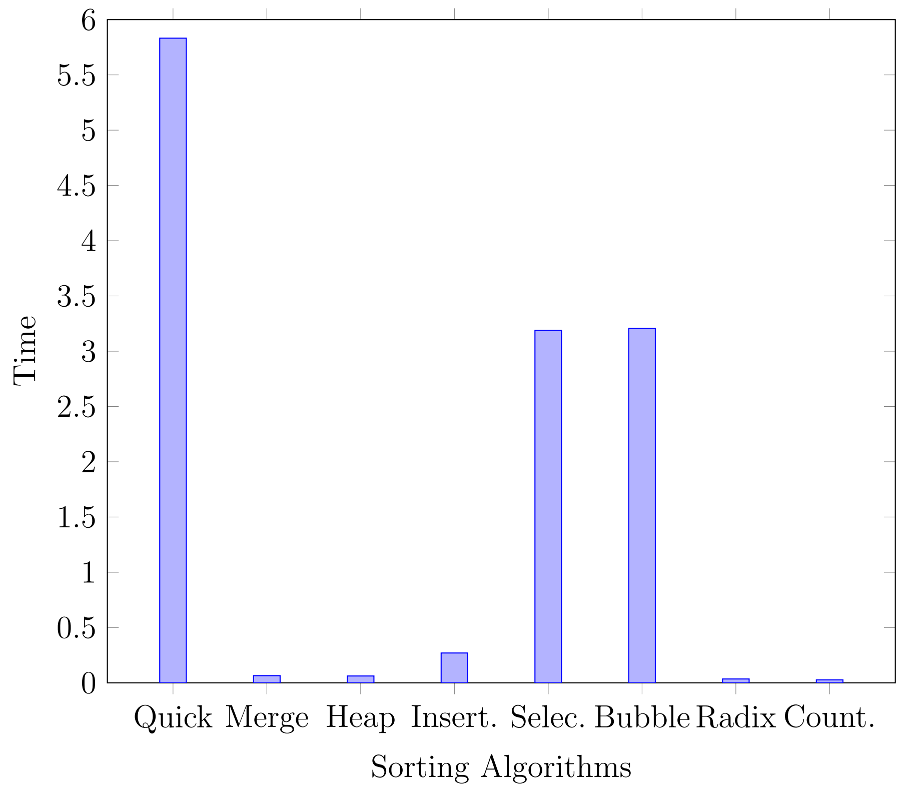
To begin the experiment I first needed the 3 arrays, which I got them from an online array generator ("https://www.onlinetools.com"). I generated those 3 arrays and I stored them in 3 different text files, while the output was shown in only one text file. I started with the array of 5000 elements, and continued with the array of 50000 elements and respectively the array of 500000 elements.

Starting with the lists of 5000, all the sorting algorithms took approximatively the same time, between 0.026s and 0.098s.



In

the next stage of the experiment, I started to sort arrays of 50000 items. Here we already started to see a difference of seconds, not only fractions of seconds. The time interval was between 0.027s and 6s.

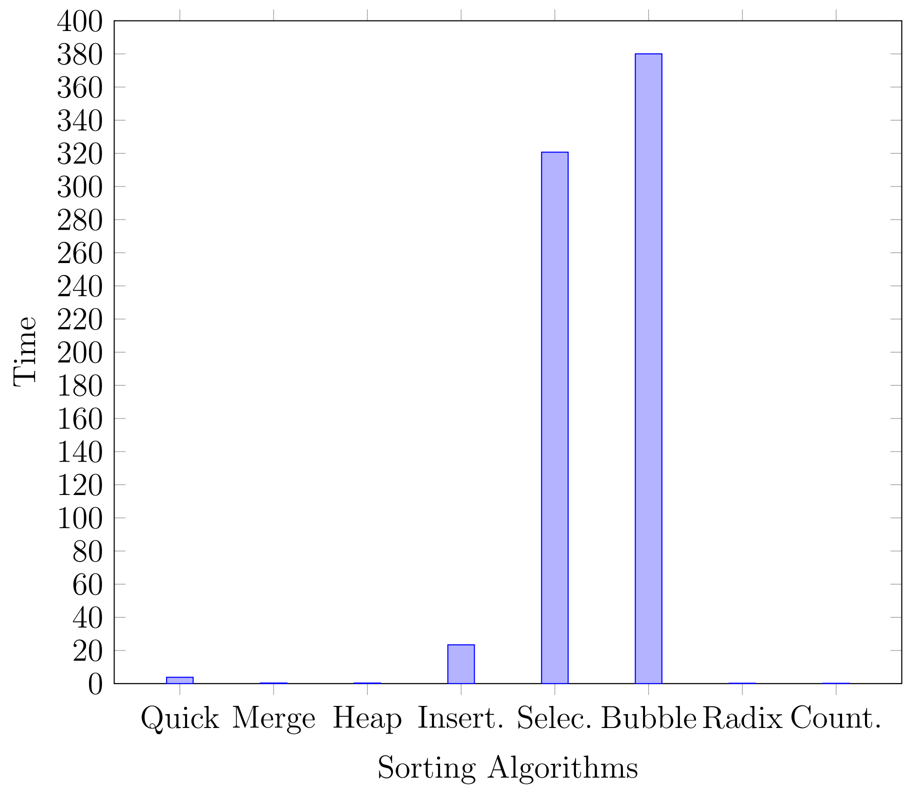


In this case we can see the that quick sort did the worst, while the best time went to Counting Sort.

Observations:

- Quick Sort did a great job sorting the random array (0.038s), but he encountered some difficulties when I started to move some items around (the first or the last).

On the next stage, things got even more interesting while I was sorting arrays with 500000 items. The time was between 0.038s and 926.835s.



Observations:

- Even if insertion sorted the random array in 162.659s and clearly worst than merge/quick/heap sort which sorted it in under 0.420s, it did the best job when I started to move some of the items around, getting a time under 0.300s, that is why the insertion average is so low, even if it took over 2 minutes to sort the entire array.
- In some cases, is better to use insertion, even on large amount of lists if there are only a few elements which are not sorted.
- Quick sort performs better on larger arrays than on smaller arrays.

- Bubble sort may be easy to use, but the efficiency is far from good, especially in large arrays.
- Depending on case, all of the above sorting algorithms are good, there is no best algorithm, it all depends on where you want to use it
- On big arrays, selection sort had a hard time trying to sort the list when the smallest value was the last and the bigger value was the first, even if the rest of the array was already sorted.

5 Related Work

The experiment we have just conducted is not something new. Many experiments and different approaches have been made by other individuals [9]. In this paper, I aim to assist readers in selecting the best algorithm for their product in terms of its complexity, adaptability, and stability. The purpose of this paper is solely to help you gain a better understanding of the advantages and disadvantages of each algorithm.

6 Conclusions and Future Work

During the process of writing this paper, I became increasingly aware of the abundance of erroneous information that exists on the internet. It was clear to me that one could easily be misled by unverified information. This realization emphasizes the importance of verifying sources and using reliable information when conducting research.

As I digged deeper into the topic of sorting algorithms, I came to the disappointing conclusion that there is no one-size-fits-all algorithm. However, there are many other variants that can meet specific needs. The most challenging aspect of this paper was sifting through the vast amount of information on sorting algorithms and extracting the essential details.

Despite these challenges, I am excited to explore and experiment with even more algorithms and to expand the area of testing to find more efficient ways of utilizing them. I hope that my research will help others navigate the world of sorting algorithms and contribute to the development of more efficient methods in the future.

References

[1] Owen Astrachan. Bubble sort: an archaeological algorithmic analysis. *ACM Sigcse Bulletin*, 35(1):1–5, 2003.

[2] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: from merge sort to timsort. 2015.

[3] Franciszek Grabowski and Dominik Strzalka. Dynamic behavior of simple insertion sort algorithm. *Fundamenta Informaticae*, 72(1-3):155–165, 2006.

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

[5] Zbigniew Marszałek. Performance test on triple heap sort algorithm. *Technical Sciences/University of Warmia and Mazury in Olsztyn*, 2017.

[6] Peter M McIlroy, Keith Bostic, and M Douglas McIlroy. Engineering radix sort. *Computing systems*, 6(1):5–27, 1993.

[7] Hadi Sutopo. Selection sorting algorithm visualization using flash. *The International Journal of Multimedia & Its Applications (IJMA)*, 3(1):22–35, 2011.

[8] Elad Verbin and Wei Yu. The streaming complexity of cycle counting, sorting by reversals, and other problems. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 11–25. SIAM, 2011.

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