A theoretical and experimental comparison of sorting algorithms

Matioc Amalia Maria
Departament of Computer Science,
West University of Timişoara,
Email: `amalia.matioc03@e-uvt.ro`

May 13, 2023

Abstract

Sorting algorithms are an important tool in computer science for organizing and manipulating large sets of data. There are many different sorting algorithms available, each with its own unique advantages and disadvantages. The challenge is to choose the right algorithm for a particular problem, depending on the size of the data set, the nature of the data, and the specific requirements of the application.

In this paper, we compare and contrast the theoretical and experimental performance of several popular sorting algorithms. We present an analysis of the time and space complexity of each algorithm, as well as their practical performance on a variety of input data sets. We also discuss the techniques used in our experiments and provide recommendations for selecting sorting algorithms based on the characteristics of the input data.

Contents

List of Tables

Code Fragments

1 Introduction

Sorting algorithms are essential tools for computer science, and efficient sorting can make a significant difference in the performance of programs that manipulate large data sets. With the vast amounts of data generated every day, efficient and accurate sorting algorithms are becoming increasingly important. In this paper, we present a theoretical and experimental comparison of several popular sorting algorithms. We will be analyzing the performance of these algorithms in both a theoretical and practical context, comparing their time and space complexity and examining their behavior on a variety of input data sets. The goal is to provide insights into the strengths and weaknesses of each algorithm and to offer guidance on selecting the best algorithm for a particular problem.

1.1 Motivation of the problem

Sorting algorithms have been studied extensively in computer science, and there are many algorithms available, each with its own unique advantages and disadvantages. However, choosing the right algorithm for a particular problem can be demanding. Existing solutions often do not provide enough guidance on which algorithm to choose, and the performance of an algorithm in a theoretical context may not reflect its practical performance on real-world data sets. As a result, it can be difficult to determine the best algorithm for a particular problem.

1.2 Informal description of solution

I will conduct an analysis and a practical experiment to study the time and space complexity of these algorithms and examine their performance across different types of input data sets as to get insights into the pros and cons when it comes to selecting the best sorting algorithm for a specific problem. In order to get to certain conclusions about the nine algorithms presented above, I have used a program in C which contains all the sorting algorithms in separate functions.

1.3 Informal example

Consider the problem of sorting a list of 100,000 integers. A straightforward solution would be to use the built-in sorting function in a programming language such as Python. However, this may not be the most efficient solution, particularly if the input data is not uniformly distributed. By comparing the performance of various sorting algorithms on different input data sets, we can identify which algorithms are best suited to specific types of data and provide guidance on selecting the optimal algorithm for a particular problem.

1.4 Declaration of originality

This paper is an independent contribution to the academic discourse on the subject matter. I have made sure to acknowledge and appropriately cite all external sources used in the research and writing process.

1.5 Reading instructions

Section 1 is the introduction. In section 2, the problem and solution will be explained in more detail. Section 3 will contain my implementation of the sorting algorithms. Section 4 will be about the results and the analysis of the experiment. In section 5, some works related to this subject matter will be referenced. Section 6 will represent the conclusions drawn from the experiment. Finally, all the references used in this research paper will be displayed at the end of the document.

2 Formal Description of Problem and Solution

In order to fully comprehend this paper I will firstly explain some of the technical terms used in this paper.

Sorting - Sorting is nothing but alphabetizing, categorizing, arranging or putting items in an ordered sequence [Akhter et al.(2016)Akhter, Idrees, et al.]. In other words, we have a sequence of numbers as input and the output will be a permutation of the input sequence such that it will be ordered based on a chosen rule.

Time complexity - The time complexity of an algorithm represents the efficiency of an algorithm defined as a function of the size of the input. It computes the time taken to execute each statement of code in an algorithm. The purpose of time complexity is to provide insight into how the execution time changes with respect to the number of operations in the algorithm, whether it increases or decreases.

We will take into consideration the time complexity of an algorithm several times in this paper due to its importance when it comes to comparing which algorithm is more suitable in a given situation.

Space complexity - The space complexity of an algorithm is the measure of the amount of memory space that an algorithm needs to solve a computational problem based on the characteristics of the input. It is the memory required by an algorithm until it executes completely.

Stability - In the context of sorting algorithms, stability refers to the property that maintains the relative order of elements with equal keys during the sorting process. More specifically, if there are two elements with equal keys in the input sequence, a stable sorting algorithm ensures that the element that appears first in the input array will also appear first in the sorted array.

Comparison-based - This is a term which we use when we refer to algorithms that sort or search for elements by comparing them pairwise.

The main problem which concerns this paper is which sorting algorithm is the best in a certain case. There exists several sorting algorithms and because there are so many it's often hard to choose one which suits one's problem. Therefore, I will conduct an experiment on nine sorting algorithms and on different data sizes: 10, 100, 10.000 and 100.000 elements. Afterwards, I will try to analyse the results and draw conclusions from them.

In this paper we will consider the following sorting algorithms:

- *Comparison based sorting algorithms:* Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Quick Sort, Merge Sort
- *Non-comparison based sorting algorithms:* Bucket Sort, Counting Sort, Radix Sort

3 Model and Implementation of Problem and Solution

In order to conduct the experimental part of this paper, I have created a C program using CodeBlocks 13.12 which contains the nine sorting algorithms, each in a separate function, another function which randomizes an array and a function which outputs the sorted array.

If you would like to visualize the full code, then it can be accessed by clicking this link.

First, I implemented a function which randomizes an array. The size of the sequence is given as input by the user.

```
    void randomize_array(int randArray[], int sz)
    {
        int i;
        for (i=0; i<sz; i++)
            randArray[i]=rand () %10000;
        for (i=0; i<sz; i++)
            printf("%d ", randArray[i]);
        printf("\n\n");
    }
```

Code Fragment 1: The function which randomizes an array

The printing function is used to output the sorted array, displaying the elements with one space between each of them.

```
    void print_array (int array[], int sz)
    {
        printf("%s \n", "The sorted array is:");
        int i;
        for (i=0; i<sz; i++)
            printf("%d ", array[i]);
    }
```

Code Fragment 2: The function which prints the array

The main function consists of taking the size of the array as input from the user and then randomizing an array based on that. In *line 8* I wrote BubbleSort because that was the sorting function I was testing, but that may be changed when using the full code by writing the name of the sorting function you would like to test.

```c
int main()
{
    int sz;
    printf("Enter the size of array:");
    scanf("%d", &sz);
    int randArray[sz], i;
    randomize_array (randArray, sz);
    BubbleSort (randArray, sz);
    print_array (randArray, sz);
    return 0;
}
```

Code Fragment 3: The main function

This analysis was conducted on a computer with an Intel Core I5 processor, 2.30 GHz and Windows 11 as its operating system. Therefore, the results could be different from one computer to another.

4 Case Studies/Experiment

In this part of the paper we will analyze the results of the experiment. First, we will analyze the comparison based algorithms in the first subsection and then the non-comparison based algorithms. After drawing conclusions from these two, the two categories will be compared.

4.1 Comparison based sorting algorithms

This type of sorting algorithms are the most common when used in practice. These algorithms use the method of comparing pairs of elements and rearranging them until the sequence is sorted in the desired order.

Name	Best case	Average case	Worst case	Space Complexity	Stable
Bubble Sort	n	n^2	n^2	1	yes
Selection Sort	n^2	n^2	n^2	1	no
Insertion Sort	n	n^2	n^2	1	yes
Heap Sort	$n \log n$	$n \log n$	$n \log n$	1	no
Quick Sort	$n \log n$	$n \log n$	n^2	$\log n$	no
Merge Sort	$n \log n$	$n \log n$	$n \log n$	n	yes

Table 1: Complexity and stability of comparison based algorithms

Based on the table above, we can notice that although Selection sort and Insertion sort have the same worst case and average case as Bubble Sort, the efficiency of the last one is worse than the other two. Despite Bubble Sort's ease of implementation, it should definitely not be used on large amounts of data, and the same goes with Selection sort and Insertion sort.

Heap sort, Quick sort and Merge sort are logarithmic algorithms, which means that they perform far better on large amounts of data as it can be seen in Table 1.

Now onto the space complexity topic, Quick sort and Merge sort have a complexity of $O(n \log n)$ and $O(n)$ due to the fact that they use additional memory space to store the temporary arrays or partitions during the sorting process, while the others require a constant amount of memory space, regardless of the input size.

When it comes to stability, we can observe that Selection Sort, Heap Sort and Quick sort are not stable. Therefore, if we need to preserve the relative order of equal elements from the input array, we should choose one of the other stable algorithms.

In general, Merge Sort has a consistent performance across all cases, with an average and worst case time complexity of $O(n \log n)$ and a space complexity of n, making it a good and stable choice for general-purpose sorting.

Name	10	100	10.000	100.000
Bubble Sort	1.330	1.364	4.180	39.163
Selection Sort	1.070	1.411	4.730	43.330
Insertion Sort	0.952	1.889	5.502	36.450
Heap Sort	0.740	1.332	3.136	15.070
Quick Sort	1.009	1.032	3.058	15.255
Merge Sort	0.783	1.413	2.639	15.450

Table 2: Running times of comparison based algorithms

From the table above we can see that Insertion sort, Selection sort and Bubble sort do fairly well on arrays up to 100 elements, but really bad on sequences of 100.000 elements.

On the other hand, the logarithmic algorithms, Heap sort, Quick sort and Merge sort seem to do better in all cases, but the most visible difference can be seen at the 100.000 elements mark, where the running times of these three algorithms are half as much as the other's.

4.2 Non-comparison based sorting algorithms

These sorting algorithms are designed to sort values without making comparisons between them. They make certain assumptions about the data, which allows for more efficient sorting techniques, hence the time complexity being linear.

Name	Best case	Average case	Worst case	Space Complexity	Stable
Bucket Sort	$n+k$	$n+k$	$n \cdot k^2$	$n+k$	yes
Counting Sort	$n+k$	$n+k$	$n+k$	$n+k$	yes
Radix Sort	$n \cdot k$	$n \cdot k$	$n \cdot k$	n	yes

Table 3: Complexity and stability of non-comparison based algorithms

Bucket Sort, Counting Sort, and Radix Sort are all non-comparison based sorting algorithms with linear time complexity, which means that they can be highly efficient for large data sets.

In terms of space complexity, Bucket Sort and Radix Sort require extra space, proportional to the input size and the range of input values. Regarding Counting sort, I think this fragment describes it best. Linear time complexity is achieved at the cost of extra space it requires to save the counting figures and execute the logic [Sandeep Kaur Gill(2019)].

An important property of these algorithms is that they are all stable.

Name	10	100	10.000	100.000
Bucket Sort	0.708	0.776	2.660	15.759
Counting Sort	0.682	0.830	3.698	15.396
Radix Sort	0.799	0.927	3.068	15.238

Table 4: Running times of non-comparison based algorithms

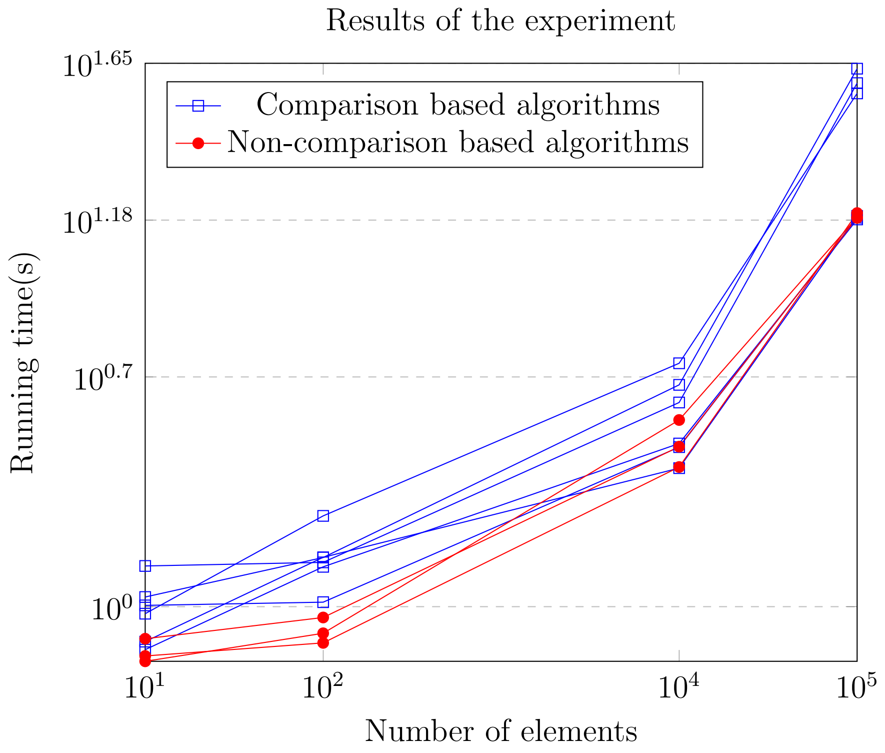
When it comes to 10 elements, Bucket sort and Radix sort behaved quite similarly. On the other hand, when we have 100.000 elements we can observe that Radix sort did the best, but Couting sort was not so far either.

It is important to note that the performance of these algorithms may vary depending on the specific characteristics of the data being sorted. For example, if the range of values in the data is relatively small, Bucket Sort or Radix Sort may perform better than Counting Sort.

While this type of algorithms are highly efficient when talking about running time, they are more difficult to implement when using other types of data rather than numbers.

4.3 Comparison vs. non-comparison based algorithms

Here is a graph containing the results of the experiment to have a better visualization of the subject matter.



We can observe that non-comparison based algorithms excelled when it comes to large data sets, but so did the logarithmic algorithms. From the experiments I have conducted I can say that non-comparison based and logarithmic sorting algorithms behaved fairly similar, while the exponential comparison based ones had higher running times in most cases.

Comparison based sorting algorithms have been compared with non-comparison sorting algorithms and concluded that they are preferred over non-comparison sorting algorithms for their simplicity when the number of values to be sorted is small, but as the size of data set increases, non-comparison sorting algorithms overtake them in terms of time complexity at the cost of extra space resources required to execute them [Sandeep Kaur Gill(2019)].

5 Related work

There have been multiple discussions based on this topic over the years and here I have listed some of them:

- Knuth, Donald E. "The analysis of algorithms." Actes du Congres International des Mathématiciens (Nice, 1970). Vol. 3. 1970.
- Bharadwaj, Ashutosh, and Shailendra Mishra. "Comparison of sorting algorithms based on input sequences." International Journal of Computer Applications 78, no. 14 (2013).
- Gill, Sandeep Kaur and Singh, Virendra Pal and Sharma, Pankaj and Kumar, Durgesh, A Comparative Study of Various Sorting Algorithms (2018). International Journal of Advanced Studies of Scientific Research, Vol. 4, No. 1, 2019.
- Al-Kharabsheh, Khalid Suleiman, et al. "Review on sorting algorithms a comparative study." International Journal of Computer Science and Security (IJCSS) 7.3 (2013): 120-126.
- Mishra, Aditya Dev, and Deepak Garg. "Selection of best sorting algorithm." International Journal of intelligent information Processing 2.2 (2008): 363-368.

6 Conclusions and Future Work

In conclusion, there is not one single "best" sorting algorithm. Some algorithms perform better on small sets of data, others on large sequences, some work better on different types of input arrays. Therefore, in order to find the best algorithm, we have to take into account the specifications of the problem, analyze them, and only after, choose the most suitable sorting algorithm for the given problem.

Further work could be done on optimizing the current ones, other sorting algorithms like Introsort, Timsort, Bead sort or parallel algorithms, and much more.

References

[Akhter et al.(2016)Akhter, Idrees, et al.] Naeem Akhter, Muhammad Idrees, et al. Sorting algorithms-a comparative study. *international Journal of Computer Science and information Security*, 14(12):930, 2016.

[Sandeep Kaur Gill(2019)] Pankaj Sharma Durgesh Kumar Sandeep Kaur Gill, Virendra Pal Singh. A comparative study of various sorting algorithms. *International Journal of Advanced Studies of Scientific Research*, 4(1), 2019.