Theoretical and Experimental Comparison of Sorting Algorithms

Mălina Mihăilescu
Departament of Computer Science,
West University Timișoara,
Email: malina.mihailescu03@e-uvt.ro

May 12, 2023

Abstract

Sorting algorithms serve as critical tools in computer science, enabling efficient data organization for a wide range of applications. This paper addresses the problem of algorithm selection and implementation by providing an analysis of different sorting algorithms. The objective is to empower programmers with insights for informed decision-making, promoting the most suitable algorithm to use considering their specific task.

The study incorporates practical experimentation conducted in the Code::Blocks IDE using C++, where the algorithms were tested on 500,000 randomly generated numbers, multiple times, allowing for the calculation of average execution times. Furthermore, to convey the performance and unique characteristics of the algorithms, the results were presented through comparative tables, graphs and analysis.

Contents

1 Introduction

Sorting algorithms are fundamental to computer science and have multiple uses across various fields. Therefore, it is crucial for both researchers and practitioners to comprehend the specifics of the different sorting algorithms and their performance characteristics. We will analyze a diverse range of sorting algorithms such as Quick Sort, Merge Sort, Bubble Sort, Insertion Sort, Selection Sort, Counting Sort, Radix Sort.

1.1 Motivation

Not all sorting algorithms are the same. Some are more efficient, some require less memory, and some are better suited for specific types of data. With so many options, it can be overwhelming to choose the right sorting method for a given task.

1.2 Solution

Through this analysis, we compare the different factors of the sorting algorithms such as time complexity, efficiency, stability and adaptability, in order to provide programmers with insights for decision-making.

1.3 Informal Example

Consider the example of a library that needs to sort a large collection of books in alphabetical order.

One existing solution the library might employ is the Insertion Sort algorithm. They could start by selecting the first book in the collection and considering it as a sorted sublist. Then, they would iterate through the remaining books, comparing each book's title to those in the sorted sublist. If a book's title is found to be alphabetically smaller, they would shift the sorted sublist to the right and insert the book in its correct position. They would continue this process until all the books are sorted. While Insertion Sort is a straightforward method, it can become time consuming and less efficient as the number of books increases.

We can consider a more efficient solution. The library could utilize the Merge Sort algorithm. Merge Sort works by dividing the collection of books into smaller subgroups, sorting them individually, and then merging the sorted subgroups back together. By repeatedly splitting the books into smaller groups and merging them in sorted order, the library can achieve a faster and more efficient sorting process.

Through this papers's analysis, valuable insights will be provided on the most suitable sorting algorithm for certain sorting needs.

1.4 Declaration of originality

The time complexities of the algorithms were taken from references [1] and [2]. The experiments, testing, analysis, conclusions, tables and graphs were done by myself. The findings presented in this paper reflect my own efforts and interpretations.

1.5 Reading Instructions

This paper starts with an introduction that emphasizes the significance of sorting algorithms as well as the issue with algorithm selection and implementation. I also provided an illustrative example to demonstrate the practical application of the sorting algorithms.

The paper contains a section for testing and experimenting, where the C++ implementations of the sorting algorithms are tested against a dataset of 500,000 randomly generated numbers, and an analysis section, where the results are put into perspective, discussing the strengths, weaknesses, and suitable use cases for each sorting algorithm.

2 Formal Description

Let us define the sorting problem formally. The objective is to rearrange a set of n elements in a specific order according to a defined criterion. In the example of sorting books alphabetically in a library, the criteria is the lexicographic order of their titles. We seek a solution that can efficiently organize the books in ascending or descending order based on this criteria.

To address the sorting problem, we propose the utilization of various sorting algorithms. The correctness of a sorting algorithm lies in its ability to rearrange the elements according to the defined criterion, ensuring that the resulting order is indeed sorted. Moreover, calculating the time complexity of the algorithm allows us to evaluate its efficiency, memory usage and potential limitations in memory constrained environments.

Let's consider the running example of sorting books alphabetically in a library. We can employ different sorting algorithms to achieve this task. For instance, Bubble Sort compares adjacent pairs of books and swaps them if they are in the wrong order, repeating this process until the entire list is sorted. Merge Sort, on the other hand, divides the book list into smaller sublists, recursively sorts them, and then merges them to obtain the final sorted list.

We would then observe that Bubble Sort may require multiple passes through the book list, while Merge Sort, with its divide-and-conquer approach, recursively splits the book list, sorts the smaller sublists, and merges them back together to obtain the sorted order in a more efficient way. The properties of correctness, time complexity, and space complexity allow us to analyze and compare these algorithms rigorously when determining their suitability for different sorting scenarios.

3 Implementation and access to code

The implementation for this project was developed using the C++ programming language within the Code::Blocks IDE. **The complete source code can be accessed through this (clickable) Github repository.**

To run the program, the code found in the main.cpp file needs to be executed in a C++ compiler.

The program follows a modular organization, where the sorting algorithm is found in a separate function and is executed in the main() function. Upon running the program, the main function automatically generates a dataset of 500,000 randomly generated numbers, from 1 to 1,000,000. These numbers serve as the input for the sorting algorithms.

The program utilizes the console interface to interact with the user. When compiled, a console window appears, where the execution time of the algorithm is displayed.

4 Experiment

To measure the execution time of the sorting algorithms, I used a laptop equipped with Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz 8GB RAM, and a 64-bit operating system with x64-based processor.

4.1 Comparison-Based Algorithms

Comparison-based algorithms are sorting algorithms that only use comparisons between elements in the input data.

4.1.1 Quick Sort

Quick sort is frequently used to sort randomized data and is thought to be one of the fastest sorting algorithms, due to its Divide and Conquer approach. It works by selecting an element from the input array as a pivot and then rearranging the array, such that all of the elements lesser than the pivot appear before the pivot, and all of the elements greater than the pivot appear after the pivot.

```
Representation on how an array gets sorted using Quick Sort:
Input array: [3, 2, 8, 1, 5]

Pivot: 5
[3, 2, 1, 5, 8]

Pivot: 8
[3, 2, 1, 5, 8]

Pivot: 1
[1, 3, 2, 5, 8]

Pivot: 3
[1, 2, 3, 5, 8]

Sorted Array: [1, 2, 3, 5, 8]
```

Quick Sort has an average and best case of O(nlogn) and a worst case of O(n^2). It's fast and efficient for large amounts of data, but it becomes inefficient for sorted data due to an unbalanced partition. The worst case of Quick Sort can be minimized by choosing the right pivot, such as the median of the unsorted array.

Upon testing the program, **the average execution time was 76 milliseconds.**

4.1.2 Merge Sort

Merge sort is an efficient Divide and Conquer algorithm. Merge sort divides a given list into smaller sublists containing only one element, and repeatedly merges those sublists until a sorted list is obtained. It uses more memory than quicksort, due to its use of temporary arrays, therefore dynamic memory allocation was used in order to be able to run the code on 500000 numbers.

```
 Representation of the Merge Sort algorithm on the array
 [5, 2, 4, 6, 1, 2, 9, 10]:

Splitting

Split the array in half:
Left half: [5, 2, 4, 6]
Right half: [1, 2, 9, 10]

Recursively splitting the subarrays

Split the left half:
Left half: [5, 2]
Right half: [4, 6]

Split the right half:
Left half: [1, 2]
Right half: [9, 10]

Merging the sorted subarrays

Merge the left half: (sorted: [2, 5]) and the right half: (sorted: [4, 6]):
Sorted subarray: [2, 4, 5, 6]

Merge the left half: (sorted: [1, 2]) and the right half:  (sorted: [9, 10]):
Sorted subarray: [1, 2, 9, 10]

Merging the final subarrays

Merge the sorted left half: [2, 4, 5, 6] and the sorted right half: [1, 2, 9, 10]:
Sorted subarray: [1, 2, 2, 4, 5, 6, 9, 10]
The final sorted array is [1, 2, 2, 4, 5, 6, 9, 10].
```

MergeSort has an average,best and worst case of O(nlogn). It's fast and efficient but requires extra memory for sorting. **The average execution time was 287 milliseconds.**

4.1.3 Bubble Sort

Bubble sort is a sorting algorithm that repeatedly compares adjacent elements in a list and swaps them if they are not in the correct order, until the entire list is sorted.

```
Representation of the Bubble Sort algorithm for
the array [6, 2, 8, 4, 10]:

Pass 1:

Compare:
6 and 2: [2, 6, 8, 4, 10]
6 and 8: [2, 6, 8, 4, 10]
8 and 4: [2, 6, 4, 8, 10]
8 and 10: [2, 6, 4, 8, 10]

Pass 2:

Compare:
2 and 6: [2, 6, 4, 8, 10]
6 and 4: [2, 4, 6, 8, 10]
6 and 8: [2, 4, 6, 8, 10]
8 and 10: [2, 4, 6, 8, 10]

The final sorted array is [2, 4, 6, 8, 10].
```

Bubble Sort has an average,best and worst case of O(n^2). It's straightforward and simple to implement, but inefficient for large sets of data. **The average execution time was 28 minutes, 54 seconds, 699 milliseconds.**

4.1.4 Insertion Sort

Insertion sort works by repeatedly taking an element from the unsorted part of the array and inserting it into the correct position in the array, until all the elements are sorted.

```
Representation of the Insertion Sort algorithm for the array [1, 5, 7, 3, 2]:


Initial array: [1, 5, 7, 3, 2]

Take the second element, 5, and compare it with the elements before it.
5>1: we leave it in its position: [1, 5, 7, 3, 2]
```

```
Take the third element, 7, and compare it with the elements before it.
7>5: we leave it in its position: [1, 5, 7, 3, 2]


Take the fourth element, 3, and compare it with the elements before it.
3<7, so we shift 7 to the right.
3<5, so we shift 5 to the right.
Insert 3 in its correct position: [1, 3, 5, 7, 2]


Take the fifth element, 2, and compare it with the elements before it.
2<7, so we shift 7 to the right.
2<5, so we shift 5 to the right.
2<3, so we shift 3 to the right.
Insert 2 in its correct position: [1, 2, 3, 5, 7]

The final sorted array is [1, 2, 3, 5, 7].
```

Insertion sort has a best case of O(n), when all the elements are sorted, and an average and worst case of O(n^2). It's efficient for smaller lists and mostly sorted lists but not for large arrays. **The average execution time was 2 minutes, 33 seconds, 692milliseconds.**

4.1.5 Selection Sort

This algorithm works by selecting the smallest element from the unsorted array and swapping it with the first element of the unsorted array.

```
Representation of the Selection Sort algorithm for the array [1, 5, 7, 3, 2]:

Step 1:

Initial array: [1, 5, 7, 3, 2]
Find the smallest element in the unsorted part of the array.
Swap it with the first element.
After the first pass, the smallest element 1 is in its correct position: [1, 5, 7, 3, 2]

Step 2:

Find the smallest element in the unsorted part of the array (excluding the first element).
Swap it with the second element.
After the second pass, the smallest element 2 is in its correct position: [1, 2, 7, 3, 5]

Step 3:
```

```
Find the smallest element in the unsorted part of the array
,excluding the first two elements.
Swap it with the third element.
After the third pass, the smallest element 3 is in its correct position: [1, 2, 3, 7, 5]

Step 4:

Find the smallest element in the unsorted part of the array
,excluding the first three elements.
Swap it with the fourth element.
After the fourth pass, the smallest element 5 is in its correct position: [1, 2, 3, 5, 7]
The final sorted array is [1, 2, 3, 5, 7].
```

Selection Sort has a best, average, and worst case of O(n^2), therefore it's slow and not suitable for large data sets. **The average execution time was 4 minutes, 57 seconds, 411 milliseconds.**

4.2 Non-Comparison-Based Algorithms

Non-comparison-based algorithms are sorting algorithms that do not use comparisons between elements in the input data.

4.2.1 Counting Sort

Counting sort works by counting the number of occurrences of each element of the array and then using that information to compute the position of each element in the sorted output array. This is not a comparison-based sorting algorithm, and instead it uses key values as indexes.

```
Representation of the Counting Sort algorithm for the input array {1, 4, 1, 1, 2, 7, 2}:


Count the occurrences of each element in the input array:
Element 1: count[1] = 3
Element 2: count[2] = 2
Element 4: count[4] = 1
Element 7: count[7] = 1


Calculate the cumulative sum of the count array:
count[1] = 3
count[2] = 5
count[4] = 6
count[7] = 7
```

```
Create an output array of the same size as the input array:
[0, 0, 0, 0, 0, 0, 0, 0]
Step 5:

From left to right:
Place elements n at index x, where x is count[n]-1.
Element 1: Place it in the output array at index 0 (count[1] - 1)
and decrement count[1]: output = [1, 0, 0, 0, 0, 0, 0, 0]
Element 4: Place it in the output array at index 3
and decrement count[4]: output = [1, 0, 0, 4, 0, 0, 0, 0]
Element 1: Place it in the output array at index 2
and decrement count[1]: output = [1, 0, 1, 4, 0, 0, 0, 0]
Element 1: Place it in the output array at index 1
and decrement count[1]: output = [1, 1, 1, 4, 0, 0, 0, 0]
Element 2: Place it in the output array at index 4
and decrement count[2]: output = [1, 1, 1, 4, 2, 0, 0, 0]
Element 7: Place it in the output array at index 6
and decrement count[7]: output = [1, 1, 1, 4, 2, 0, 7, 0]
Element 2: Place it in the output array at index 5
and decrement count[2]: output = [1, 1, 1, 4, 2, 2, 7, 0]


The output array is the sorted array:
[1, 1, 1, 2, 2, 4, 7]
```

Counting-sort runs in O(n) time and it's an efficient sorting algorithm with linear running time. **The average execution time was 12 milliseconds.**

4.2.2 Radix Sort

The Radix sort algorithm works by sorting the numbers digit by digit, from the LSD (least significant digit) to the MSD (most significant digit) or from the MSD to the LSD. This is a non-comparison based algorithm, and instead it groups numbers into "buckets" according to their digit.

```
Representation of Radix Sort for the
input array [34, 12, 89, 45, 27,81,29].

We will create ten buckets (0 to 9) to hold the numbers based on their digits.

Pass 1: Based on the rightmost digit (1's place)
The numbers are placed in the buckets according to their rightmost digit.
Bucket 0:
Bucket 1: 81
Bucket 2: 12
```

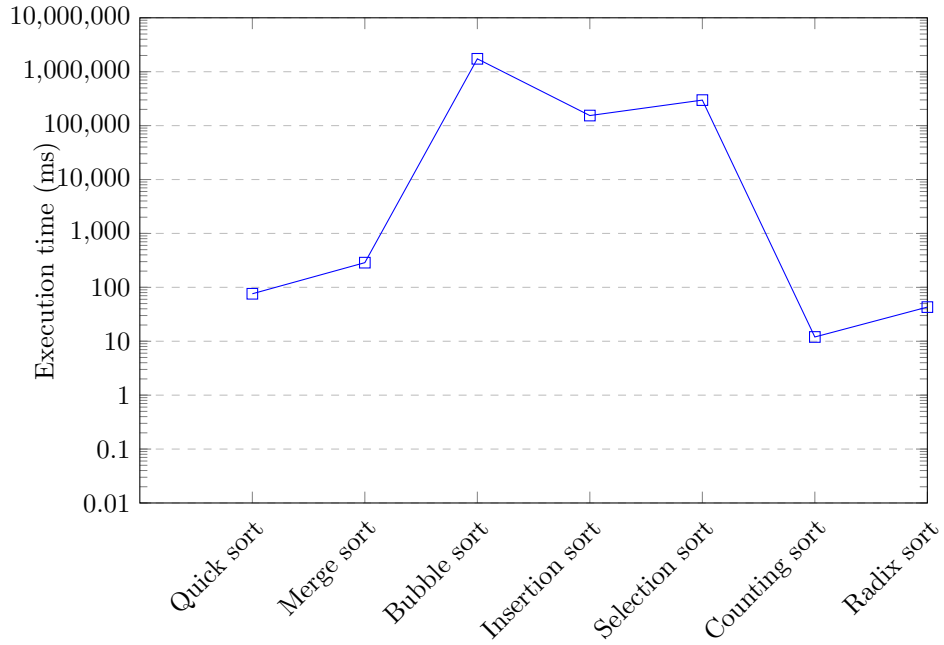
```
Bucket 3:
Bucket 4: 34
Bucket 5: 45
Bucket 6:
Bucket 7: 27
Bucket 8:
Bucket 9: 89 , 29

[81, 12, 34, 45, 27, 89, 29]

Pass 2: Based on the second rightmost digit (10's place)
The numbers are placed in the buckets according to their second rightmost digit.

Bucket 0:
Bucket 1: 12
Bucket 2: 27 , 29
Bucket 3: 34
Bucket 4: 45
Bucket 5:
Bucket 6:
Bucket 7:
Bucket 8: 81, 89
Bucket 9:

Sorted array: [12, 27, 29, 34, 45, 81, 89]
```

Radix Sort has a time complexity of (d(n + k)), where d is the number of digits in the largest integer and each digit can take k values. [3, pg 3]. **The average execution time was 43 milliseconds.**

5 Analysis

Sorting Algorithm	**Execution Time (ms)**
Quick sort	76
Merge sort	287
Bubble sort	1734699
Insertion sort	153692
Selection sort	297411
Counting sort	12
Radix sort	43

Table 1: Comparison of Execution Time for Different Sorting Algorithms



As we can see from the graphs, Quick sort, Merge sort, and non-comparison based algorithms like Counting sort and Radix sort have superior efficiency when it comes to sorting large amounts of data, with Counting sort being the fastest one. On the other hand, Bubble sort appears to be disproportionately inefficient compared to the other algorithms, having the longest execution time. Insertion sort and Selection sort have an intermediate performance in terms of runtime.

It's important to keep in mind, however, that different sorting algorithms may be used under different circumstances, depending on the type of data, the size of the dataset, and the requirements of the task. Sorting algorithms have their advantages and disadvantages, which can make them better suited for certain applications.

For example, Quick sort and Merge sort are both efficient algorithms and are commonly used. Regardless, Quick sort is preferred for smaller arrays, it's unstable, since it does not guarantee the input order of equal elements, and it requires less memory, while Merge sort is very efficient at sorting linked lists, operates fine on any size of array, is stable, but typically requires additional memory allocation.

Bubble sort is stable, easy to implement, and doesn't require additional memory space, but it's very slow and inefficient when working with large datasets.

Insertion sort is more efficient than Selection sort, it's stable and usually used on small arrays or partially sorted arrays since it can skip over the elements that are already in the correct order. Insertion sort does not require any additional memory allocation beyond the input array. In contrast, Selection sort often

requires additional memory to store the temporary minimum element during each step of the sorting process.

Radix sort has a linear time complexity, performing faster than comparison-based algorithms for larger datasets. However, it requires extra memory to store the intermediate results of sorting the elements by each digit, therefore it might be less memory efficient than in-place sorting algorithms.

Counting sort is also a non-comparison based sorting algorithm with linear complexity, exhibiting better performance than comparison-based algorithms like Quick sort or Merge sort. It might require additional memory to store the counts for each element, especially when working with a large range of values.

6 Future work and conclusion

In this paper, I addressed the problem of analyzing sorting algorithms and providing insights for informed algorithm selection. The main objectives were to assess the performance of different sorting algorithms and provide recommendations for their appropriate usage.

During the implementation and analysis, I encountered difficulties related to preventing memory overflow for algorithms that require more memory. Some sorting algorithms, such as Merge Sort, have a higher memory requirement due to their divide-and-conquer nature. To address this challenge, I employed dynamic memory allocation techniques.

While I achieved my primary objectives in this study, there are still several avenues for future work. Firstly, exploring ways to further improve the efficiency of the sorting algorithms would be valuable. This could involve investigating alternative data structures, or exploring hybrid algorithms that combine the strengths of different approaches.

In addition, it would be beneficial to update the algorithms to handle worst-case scenarios, as they may perform poorly in specific instances that could otherwise be avoided.

In conclusion, this paper provides an analysis of sorting algorithms and their unique characteristics, so that practitioners can select the best sorting algorithm for their specific needs. Further research and innovation in this field can ultimately contribute to the development of increasingly efficient data organization solutions, enabling better performance and productivity in various applications.

References

[1] Mishra, Aditya Dev and Garg, Deepak, "Selection of best sorting algorithm," *International Journal of Intelligent Information Processing*, vol. 2, no. 2, pp. 363-368, 2008.

[2] Prajapati, Purvi and Bhatt, Nikita and Bhatt, Nirav, "Performance comparison of different sorting algorithms," *Vol. VI, no. VI*, pp. 39-41, 2017.

[3] P. Horsmalahti, *Comparison of bucket sort and radix sort*, arXiv preprint arXiv:1206.3511, 2012.

[4] Khalid Suleiman Al-Kharabsheh, Ibrahim Mahmoud AlTurani, Abdallah Mahmoud Ibrahim AlTurani, and Nabeel Imhammed Zanoon. *Review on sorting algorithms: a comparative study. International Journal of Computer Science and Security (IJCSS)*, 7(3):120–126, 2013.

[5] Vladmir Estivill-Castro and Derick Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 441–476, 1992.