A Theoretical and Experimental Comparison of Sorting Algorithms

Arina Teodorescu
West University of Timişoara, Romania

May 2024

Abstract

In this study, I analyze the key characteristics and performance of several well-known sorting algorithms, including but not limited to bubble sort, selection sort, insertion sort, merge sort, and quick sort. I conduct experimental evaluations to compare their time complexity, space complexity, and overall efficiency in sorting different types and sizes of datasets. The results of this study provide valuable guidance for practitioners in selecting the most appropriate sorting algorithm based on the specific constraints and requirements of their applications.

Contents

1 Introduction

Sorting algorithms are fundamental to computer science and play a crucial role in various applications. We will explore the significance of sorting algorithms, discuss different types of sorting techniques, and analyze their performance characteristics.

1.1 Motivation of the Problem

Choosing the right sorting algorithm is crucial for efficient data processing. Different algorithms have varying time complexities and space requirements, making them suitable for different types and sizes of data. The selection can directly impact system performance and scalability.

For example, quick sort and merge sort are efficient for large datasets, while insertion sort and bubble sort may be more suitable for small datasets. Understanding the advantages and limitations of each sorting algorithm is essential to optimize efficiency in data processing operations.

1.2 Informal Description of Solution

The paper aims to provide a detailed comparison of well-known sorting algorithms, including bubble sort, selection sort, insertion sort, merge sort, and quick sort. By conducting theoretical and experimental analyses, the goal is to evaluate their time complexity and overall efficiency. Through this process offers insights into the strengths and weaknesses of these algorithms can enable informed decision-making when choosing the most suitable algorithm for specific use cases.

1.3 Informal Example

Consider a scenario where a programmer needs to sort a large dataset of customer transactions. The choice of sorting algorithm could significantly impact the processing time and resource consumption. This study seeks to illustrate the performance of different sorting algorithms in real-world scenarios, offering practical guidance for individuals tackling similar challenges.

1.4 Reading Instructions

This paper is organized as follows:

2 Formal Description of Problem and Solution

2.1 Problem

The sorting problem can be formally defined as follows: Given a sequence of n elements $[a_1, a_2, \ldots, a_n]$, the goal is to rearrange the elements in non-decreasing order. The input sequence may consist of randomly generated or nearly sorted lists.

2.2 Solution

The solution involves a detailed comparison of well-known sorting algorithms, namely bubble sort, selection sort, insertion sort, merge sort, quick sort, radix sort, heap sort and counting sort. The algorithms are analyzed theoretically and experimentally to evaluate their performance characteristics.

2.2.1 Bubble Sort

- **Definition & Mechanism**: Bubble sort is a simple algorithm that repeatedly compares adjacent list elements and swaps them if needed, until the list is sorted.
- **Complexity**: It has a worst case and average time complexity of O(n^2), improving to O(n) for already sorted lists. Its space complexity is O(1).
- **Advantages**: Bubble sort is easy to understand and implement, memory efficient and useful for small or nearly sorted datasets.
- **Disadvantages**: It is less efficient for larger lists compared to other algorithms like quick sort, merge sort, and heap sort.

2.2.2 Merge Sort

- **Definition & Mechanism**: Merge sort is a divide-and-conquer algorithm that recursively divides the input, sorts each half, and merges them back together.

- **Complexity**: Merge sort has a consistent time complexity of O($n \log n$) and a worst case space complexity of O(n) due to its use of temporary arrays for merging.
- **Advantages**: It is efficient and works well with large datasets and non-random access data structures. It's also a stable sort algorithm.
- **Disadvantages**: Merge sort requires additional memory for temporary arrays equivalent to the input size, and its implementation is more complex than simpler algorithms.

2.2.3 Quick Sort

- **Definition & Mechanism**: Quick sort is an efficient divide-and-conquer algorithm. It selects a 'pivot' element, partitions the array into two sub-arrays based on this pivot, and repeats the process recursively.
- **Complexity**: Quick sort has a best and average time complexity of O($n \log n$), but can degrade to O(n^2) in the worst case. The space complexity is O($\log n$).
- **Advantages**: Quick sort is faster than many sorting algorithms like Bubble Sort and Insertion Sort for larger datasets. It doesn't require extra storage as it's an in-place algorithm.
- **Disadvantages**: It is not a stable sort, and the O(n^2) worst case time complexity can occur with a poor pivot selection.

2.2.4 Insertion Sort

- **Definition & Mechanism**: Insertion sort is a comparison-based sorting algorithm that builds a sorted array one item at a time, akin to sorting playing cards in your hands.
- **Complexity**: It has a best case time complexity of O(n), but is O(n^2) in average and worst case scenarios. The space complexity is O(1).
- **Advantages**: It's simple to implement, efficient for small or nearly sorted datasets, it's a stable sort and it doesn't require additional memory.
- **Disadvantages**: Insertion sort is not suitable for larger datasets due to its O(n^2) time complexity, and it's less efficient than sorts like quick sort, merge sort and heap sort.

2.2.5 Selection Sort

- **Definition & Mechanism**: Selection sort is a comparison-based algorithm that divides the input into sorted and unsorted regions, finds the smallest (or largest) element in the unsorted region, and swaps it with the first unsorted element.
- **Complexity**: Both the best and worst case time complexity is $O(n^2)$, and the space complexity is $O(1)$.
- **Advantages**: It is suitable for small lists, efficient when swapping items is relatively cheap, stable, and makes the minimum number of swaps.
- **Disadvantages**: It is less efficient for larger lists and compared to other algorithms like quick sort, merge sort, and heap sort.

2.2.6 Heap Sort

- **Definition & Mechanism**: Heap sort is a sorting algorithm that uses a binary heap data structure. It iteratively extracts the largest element and moves it to the sorted region.
- **Complexity**: Its time complexity is consistently $O(n \log n)$ for all scenarios, and the space complexity is $O(1)$.
- **Advantages**: Heap sort is efficient and performance is not affected by the initial order of elements, it has better worst-case runtime than quick sort and does not require additional storage.
- **Disadvantages**: Heap sort is not a stable sort, while theoretically efficient, it may lag in practice due to structural reasons and may exhibit poor cache performance.

2.2.7 Radix Sort

- **Definition & Mechanism**: Radix sort is a non-comparative integer sorting algorithm. It sorts numbers by grouping digits that are in the same position and have the same value, starting from the least significant.
- **Complexity**: The time complexity is generally $O(d * (n + b))$, while the space complexity is $O(n + b)$, where n is the number of items, d is the number of digits, and b is the base of the represented number.
- **Advantages**: It can outperform comparison-based sorts like quick sort or merge sort when the length of the input is significant and is useful for sorting items represented digitally like texts or images.
- **Disadvantages**: Radix sort is not adaptive and less efficient for datasets with floating-point numbers or strings.

2.2.8 Counting Sort

- **Definition & Mechanism**: Counting sort is a sorting algorithm that counts the number of elements for distinct key values. It is used when the range of input values is small.
- **Complexity**: Its time complexity is $O(n + k)$, and the space complexity is also $O(n + k)$, where n is the number of elements and k is the range of input.
- **Advantages**: It maintains the relative order of equal elements (stable) and performs the sorting process in linear time regardless of data distribution.
- **Disadvantages**: It's not suited for large ranges due to high space complexity and it solely works with lists of integer values due to its non-comparison based nature.

2.3 Properties of Solution

In evaluating my solution, I focus on two key aspects: ensuring its correctness and optimizing its efficiency.

Each sorting algorithm implemented in my solution follows accepted principles and methodologies, ensuring their theoretical correctness. For instance, bubble sort repeatedly traverses the list and swaps adjacent elements if they are in the wrong order. Merge sort divides the list into two halves, sorts them, and merges them. The well-defined nature of these algorithms provides confidence in their ability to perform accurate sorting regardless of the input conditions.

The solution's efficiency hinges on the sorting algorithm. Some, like quick sort, excel for large datasets but can be slowed by poor pivot choices. Others, like selection sort, handle small or nearly sorted data well, but struggle with large sets. To tackle this, I've incorporated a variety of sorting algorithms into my solution, allowing me to pick the most efficient tool for each specific scenario.

3 Model and Implementation of Problem and Solution

3.1 Problem Modeling

The problem is represented in the form of multiple lists of integers, which will be sorted using various algorithms. These lists are generated in two specific

ways:

- Random: Lists are populated with randomly generated integers.
- Nearly Sorted: Lists are mostly sorted but have a small percentage of elements out of place.

In both these categories, list sizes vary to test the performance of sorting algorithms against different volumes of data.

3.2 Solution Implementation

The solution is built in Python 3.9.6, implementing eight distinct sorting algorithms: quick, merge, radix, heap, insertion, bubble, selection and counting sort and a few additional ones for generating data. Each sorting function can operate independently, allowing me to call on them one by one as needed.

In constructing this project, I used special libraries such as *random* to generate data, *heapq* for heap sort, and *timeit* to measure how long each sorting algorithm took. These libraries are part of Python's standard library, so there's no need for additional downloads.

4 Experiment

4.1 Objective

The objective for this experiment is to analyze and compare the efficiency of the previously mentioned sorting algorithms. The tests were executed on lists of different sizes (100, 500, 1000, 5000 items), that were either randomly populated or nearly sorted, with 5% of items swapped.

4.2 Setup

For the experiment setup, I created a list for each category (random and nearly sorted) and each size under question. Afterwards, I executed each sorting algorithm on these lists, timing each execution to evaluate performance. Timing is done using the *timeit* module in Python. In the interest of fair comparison, I made sure to use deep copies of the list for each individual sorting algorithm.

4.3 Results

The results of my experiment are summarized in the table below:

Table 1: Execution times of sorting algorithms for different scenarios

Scenario	Algorithm	Execution Time (s)
100 (Random)	heap sort	0.0018
	radix sort	0.0050
	bubble sort	0.0346
	merge sort	0.0104
	quick sort	0.0075
	insertion sort	0.0158
	selection sort	0.0159
	counting sort	0.0098
100 (Nearly Sorted)	counting sort	0.0018
	heap sort	0.0016
	bubble sort	0.0197
	merge sort	0.0097
	quick sort	0.0066
	insertion sort	0.0031
	selection sort	0.0154
	radix sort	0.0031
500 (Random)	heap sort	0.0100
	counting sort	0.0138
	bubble sort	0.8691
	merge sort	0.0652
	quick sort	0.0455
	insertion sort	0.4202
	selection sort	0.4109
	radix sort	0.0224
500 (Nearly Sorted)	heap sort	0.0091
	counting sort	0.0090
	bubble sort	0.4940
	merge sort	0.0617
	quick sort	0.0388
	insertion sort	0.0535
	selection sort	0.4098
	radix sort	0.0221
1000 (Random)	heap sort	0.0209
	counting sort	0.0184
	bubble sort	3.7349
	merge sort	0.1439
	quick sort	0.0933

Continued on next page

Table 1 – continued from previous page

	insertion sort	1.7379
	selection sort	1.6332
	radix sort	0.0448
1000 (Nearly Sorted)	heap sort	0.0191
	counting sort	0.0184
	bubble sort	2.1564
	merge sort	0.1330
	quick sort	0.0794
	insertion sort	0.2043
	selection sort	1.6290
	radix sort	0.0444
5000 (Random)	counting sort	0.0474
	heap sort	0.1226
	bubble sort	98.5610
	merge sort	0.8698
	quick sort	0.3734
	insertion sort	44.8263
	selection sort	40.3288
	radix sort	0.2224
5000 (Nearly Sorted)	counting sort	0.0920
	heap sort	0.1080
	bubble sort	57.9273
	merge sort	0.8033
	quick sort	0.4477
	insertion sort	5.1725
	selection sort	39.7471
	radix sort	0.2872

Upon reviewing the results, bubble sort was the least effective, particularly with larger lists. Radix, heap, quick, and merge sort showed impressive performance on larger and nearly sorted lists. Notably, insertion sort performed well on nearly sorted lists despite its overall insufficient results.

These observations echo the theoretical time complexities of these respective algorithms. Bubble sort, insertion sort, and selection sort, which exhibit quadratic time complexity $O(n^2)$, understandably perform worse for larger datasets. Comparatively, heap sort, merge sort, and quick sort, with time complexity of $O(n \log n)$, perform much better when confronted with voluminous inputs.

4.4 Reproducing the Experiment

To reproduce this experiment, clone the code from my GitHub repository `https://github.com/AarpadD/Activity-1_mpi`, ensure you have Python 3.9.6 installed on your system and run the script per the instructions provided. The script will output the execution times for each algorithm over different list sizes and types.

5 Conclusion

To sum things up, I compared different sorting algorithms in this study both through theory and actual tests.

The tests showed that the speed of sorting algorithms can differ a lot based on the type of data they're working on. For instance, merge sort and heap sort performed well on both random and nearly sorted lists. This means they are reliable for handling differing kinds of data. On the other hand, bubble sort and insertion sort didn't perform as well, particularly with bigger lists of data, and this was consistent with their time complexity.

Looking at the theory, it explains why some algorithms work faster than others. Bubble sort and insertion sort have a time complexity of $O(n^2)$, which means their speed reduces a lot as the amount of data increases. On the other end, merge sort and heap sort have a time complexity of $O(n \log n)$, meaning they handle large data sets better.

Furthermore, radix sort and counting sort highlight that even the specifics of the data can shape an algorithm's performance. Radix sort is great for specific amounts of digits and counting sort works well with smaller groups of positive numbers. So, it's all about picking the right algorithm for the job.

In conclusion, understanding these algorithms means weighing the pros and cons of each one. A good grasp of the theory, coupled with testing them out in act, is key to making a good decision.

6 References

- A. K. Karunanithi (2014). A Survey, Discussion and Comparison of Sorting Algorithms
  https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d010950f6b3c9521eb437334fa69c0b2b9353010
- D.T.V Dharmajee Rao, B.Ramesh (2012). Experimental Based Selection of Best Sorting Algorithm
  https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=295ee11e71ff42e4d74dbc83d320d462d4695e67