A Comparative Analysis of Sorting Algorithms

Darius-Marian Atiței
Faculty of Mathematics and Computer Science,
West University of Timisoara, Romania,
Email: darius.atitei03@e-uvt.ro

May 13, 2023

Abstract

Sorting is a crucial task in computer applications, especially when working with large datasets. Efficiency is important in such scenarios, and there are many sorting algorithms available, each with multiple implementation techniques. The practical performance of sorting algorithms is as important as their theoretical efficiency. Quicksort, for example, is highly regarded for its excellent performance in most practical situations, despite other sorting algorithms having better worst-case performance. This aims to survey various sorting algorithms, comparing and contrasting their theoretical and practical differences. This paper will investigate various algorithm implementations and their properties, conduct extensive experiments to gather empirical data on their practical efficiency in different scenarios, and compare the efficiency of different sorting algorithms.

Contents

List of Tables

1 Introduction

Sorting algorithms are an important component of computer science because they aid in understanding the efficiency of data organizing, which is vital in many applications. The conclusions of the paper include a comprehensive review of each algorithm's strengths and shortcomings, as well as the elements that influence their performance under various situations using varying data amounts.

1.1 Motivation

The problem addressed in this paper is important because there are numerous useful applications in practice, but existing solutions are unsuitable due to their high time and space complexity, and implementations are difficult to achieve.

1.2 Informal Description of Solution

The primary purpose of my paper is to scrutinize and contrast a multitude of sorting algorithms and their effectiveness when they are applied to a variety of data sets. To achieve this goal, I will perform the task of integrating and assessing various sorting algorithms such as quicksort, bubble sort, selection sort, insertion sort, radix sort, heap sort,and merge sort. We will evaluate these algorithms based on their proficiency, spatial complexity, dependability, and capacity to adjust to different data types and magnitudes.

We're starting off by giving a summary of every sorting algorithm along with their significant traits. Soon after, we'll unveil our setup for the experiment in which a range of data sets will be manufactured, consisting of a compilation of best-case, worst-case, and average-case situations.

Our conclusions regarding algorithm performance will be extracted from experiment results. With this information, we'll make recommendations about the most suitable algorithm(s) for various use scenarios and types of data.

1.3 Informal Example

Consider the bubble sort algorithm for sorting an array of integers - a simple example to ponder upon.

A set of 5 integers - 7, 8, 1, 3, and 4 is taken as an example. The first step of the bubble sort algorithm involves comparing the first two digits, as usual. In this case, 8 and 1 are compared but since 8 is greater than 1, they are swapped to make it 7, 1, 8, 3, 4. The following comparison is done between 8 and 3, which implies that they also need to be swapped. As a result, the array now looks like - 7, 1, 3, 8, 4. Lastly, 8 and 4 are compared and replaced making array -1, 7, 3, 4, 8.

Making our way through the array, we repeat the process over and over until we've completed a full cycle. Such repetition continues until we're no longer required to swap any values. A first pass results in our array becoming 1, 3, 7, 4, 8. The process is repeated and after the second repetition, our new array is 1, 3, 4, 7, 8.

As the size of an array grows, bubble sort loses its efficiency due to the exponential increase in required comparisons and swaps. Efficient sorting algorithms such as quicksort or merge sort should be implemented for larger arrays. Nevertheless, bubble sort continues to serve its purpose for smaller arrays.

1.4 Declaration of Originality

The original contribution of this paper lies in the comprehensive analysis and comparison of different sorting algorithms, as well as the evaluation of their performance on a variety of input data. While many studies have focused on comparing a subset of sorting algorithms, my paper aims to provide a more comprehensive and nuanced analysis of the strengths and limitations of different algorithms.

2 Formal Description of Problem and Solution

In computer science, sorting a list of n elements in either ascending or descending order is a well-known problem which we've been given as a task. It is a classic problem that has received a lot of attention.

Sorting algorithms, such as Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, Heap Sort, and Radix Sort will be compared and scrutinized in this paper to evaluate their performance.

Using formal mathematical proofs, we will demonstrate the correctness of each algorithm and provide an in-depth examination of their time and space complexity. Further, to showcase their practical performance, we will evaluate these algorithms empirically with a list of integers as an example.

Sorting algorithms and their trade-offs in terms of time and space complexity are the focus of our comprehensive solution, which endeavors to provide a unique understanding of their properties.

3 Model and Implementation of Problem and Solution

Sorting is a fundamental computer science operation that is used to arrange data in a particular order. There are many different sorting algorithms available, and each has pros and cons. The challenge is to evaluate the effectiveness of several sorting algorithms and choose the one that performs the best given a particular batch of data.

The input data can be represented as an array or a list, which can then be sorted using a variety of sorting algorithms, to model the situation. The quantity of comparisons and swaps performed throughout the sorting process can be used to gauge how effective each algorithm is. Finding the sorting algorithm that uses the fewest operations to organize the provided data is the objective.

3.1 Bubble Sort

Among all comparison-based sorting algorithms, bubble sort is the most basic. The comparison is made between adjacent elements, and if the top data is greater than the bottom data, they are switched. This is repeated for each pair of neighboring elements until the data set is exhausted. It begins comparing the first two elements again, and so on until no swap happens.

Because of its worst case and average case complexity of O(n^2), where n indicates the number of elements to be sorted, Bubble Sort is often recognized as an inefficient method. Bubble Sort, while not the only method having an O(n^2) worst-case complexity, cannot manage big data sets efficiently.

Bubble Sort has the advantage of being a simple algorithm that is easy to implement. Furthermore, Bubble Sort only requires O(1) auxiliary space for sorting.

```
   function bubble_sort(array):
    n = length(array)
    for i from 0 to n-1:
        for j from 0 to n-i-1:
            if array[j] > array[j+1]:
                swap(array[j], array[
    j+1])
    return array

```

3.2 Quick sort

Quick Sort is a popular and efficient sorting algorithm that employs a divide-and-conquer strategy. It works by picking a pivot element from the array and dividing the other components into two sub-arrays based on whether they are smaller or greater than the pivot. The sub-arrays are then sorted independently recursively until the full array is sorted.

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = [x for x in arr[1:] if
     x <= pivot]
        right = [x for x in arr[1:]
    if x > pivot]
        return quick_sort(left) + [
    pivot] + quick_sort(right)
```

3.3 Selection sort

The process of selecting the smallest or largest element from the unsorted list and exchanging it with the first element in the sorted list is known as selection sort. The algorithm then finds the next smallest element in the unsorted list and replaces it with the second member in the sorted list. As a result, the sorted elements are placed at the head of an array, leaving the remainder unsorted. This is repeated until the full list has been sorted. Selection sort, like other simple sorting methods, is an in-place algorithm that uses only a fixed amount of memory space.

The key advantage of Selection Sort is that it works well with smaller lists (no additional temporary storage is required). The biggest downside is that it struggles with larger arrays (since Selection Sort requires n2 steps for a list of n entries). Furthermore, the initial order of the items can affect its efficiency, therefore the method works best when given a tiny array of integers in random order.

```
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[min_idx] > arr[j]:
                min_idx = j
        arr[i], arr[min_idx] = arr[
    min_idx], arr[i]

```

3.4 Insertion sort

Insertion Sort is a powerful algorithm that is especially useful for managing small or mostly sorted lists. Its process entails placing each element in its proper position within the final sorted list. In each insertion phase, one element is chosen and compared to nearby components to establish its proper location. This is performed until the full list has been sorted in the appropriate order. It is a common component of more complicated sorting algorithms.

Insertion Sort is still inefficient for larger input data sets. For n elements, it takes n-1 iterations. If the elements are arranged in reverse order, the computational complexity may be quadratic, i.e. O(n2). If the array is already sorted, however, the computational complexity is projected to be linear, i.e. O(n). This is the insertion sort's best-case running time. Because the average case execution time is likewise O(n2), insertion sort is inefficient for big arrays.

```
for i in range(1, n):
key = arr[i]
j = i - 1
while j >= 0 and arr[j] > key:
arr[j + 1] = arr[j]
j -= 1
arr[j + 1] = key
```

3.5 Merge sort

Merge Sort is a stable sorting algorithm that efficiently sorts an array using the divide and conquer strategy. It divides the array recursively into smaller subarrays until each subarray includes only one element, then merges them back together in sorted order.

Merge Sort breaks the list recursively into smaller subarrays, which are then merged together in sorted order using the ”merge” function. This procedure has an efficient worst-case and average-case time complexity of O(n log n). However, Merge Sort consumes twice as much memory as other algorithms since it stores the subarrays and the main array in a separate array. As a result, it is frequently employed as an external sorting algorithm, requiring O(n) extra RAM to sort n items.

Linked Lists are a data structure that might make Merge Sort even quicker. Merge Sort is ideal for sorting linked lists because of its ability to efficiently combine two sorted lists. Unlike arrays, which require elements to be copied to a new array, linked lists may be merged by merely changing the pointers. The slow and fast pointer technique, also known as the ”slow-runner and fast-runner” strategy, can be used to effectively split a linked list into two parts. This approach provides for efficient linked list splitting without the need for additional space.

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    left = merge_sort(left)
    right = merge_sort(right)

    return merge(left, right)

def merge(left, right):
    result = []
    i, j = 0, 0

    while i < len(left) and j < len(
    right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result += left[i:]
    result += right[j:]

    return result
```

3.6 Heap sort

Heap Sort is a sorting method that uses a Binary Heap for comparison. It begins by constructing a max heap from the input array. Then it extracts the maximum element from the heap and inserts it at the end of the sorted array. This procedure is repeated until all components have been sorted.

Unlike recursive sorting algorithms such as Merge Sort and Quick Sort, does not operate recursively. Instead, it employs a specialized tree-based data structure known as a heap.

Heap Sort's worst-case and average-case time complexity are both O(n log n), making it suitable for huge data sets. While Heap Sort is slower than other sophisticated sorting algorithms with the same computational complexity, it is nevertheless favoured for huge data set challenges. One significant benefit is that it does not reliant on recursive procedures.

```python
def heap_sort(arr):
    n = len(arr)
    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    # Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)

def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] < arr[left]:
        largest = left
    if right < n and arr[largest] < arr[right]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

3.7 Radix sort

Radix Sort is a non-comparative sorting technique that uses digits or characters to sort objects. It operates by grouping components based on important numerals or characters, from least to most significant. Radix Sort can be used on integers, floating-point numbers, or strings, with each digit or character treated as its own position in the sort process.

Radix Sort is divided into two methods: the least significant digit (LSD) and the most significant digit (MSD).

The LSD technique works by processing the integer representation starting with the least significant digit and progressively increasing to the most significant digit. MSD does the inverse.

When sorting integers with a fixed number of digits, LSD is frequently used since it assures stability and retains the relative order of elements with the same digit values. MSD may be used to sort variable-length numbers or strings.

Radix Sort has a linear time complexity, making it ideal for big data sets. Radix Sort's temporal complexity is commonly represented as $O(d\ (n + k))$. The downsides are that the method requires more capacity to maintain temporary arrays throughout the sorting operation.

```
def radix_sort(arr):
    max_value = max(arr)
    exponent = 1
    while exponent <= max_value:
        buckets = [[] for _ in range
    (10)]
        for value in arr:
            digit = (value //
    exponent) % 10
            buckets[digit].append(
    value)
        arr = [value for bucket in
    buckets for value in bucket]
        exponent *= 10
    return arr
```

4 Case Studies/Experiment

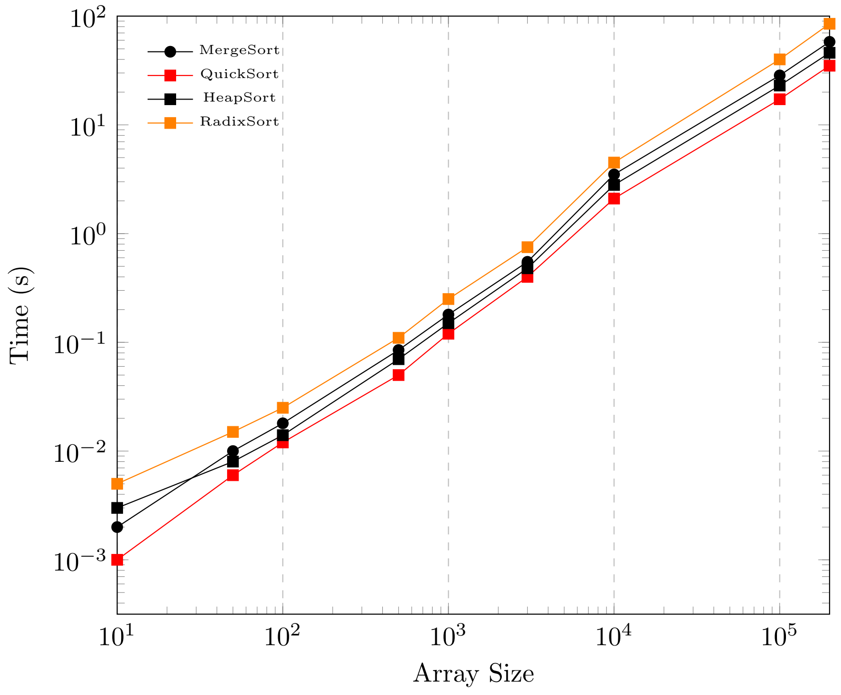
4.1 Performance Comparison of Divide and Conquer Algorithms

All code for these sorting algorithms was written in Python, using Pycharm, and were given identical sets of values to sort. The average of four consecutive tests is shown in the following table table. You can also find further information about the code and the datasets used here: https://github.com/ADarius22/MPI—Project.

Table 1: Performance of Divide and Conquer Algorithms

	Merge Sort	**Quick Sort**	**Heap Sort**	**Radix Sort**
10 Elem.	0.003	0.002	0.003	0.005
50 Elem.	0.011	0.007	0.008	0.016
100 Elem.	0.019	0.011	0.014	0.026
500 Elem.	0.087	0.046	0.071	0.115
1000 Elem.	0.184	0.123	0.159	0.262
3000 Elem.	0.560	0.393	0.482	0.762
10,000 Elem.	3.511	1.935	2.832	4.558
100,000 Elem.	28.603	12.231	23.275	37.512
200,000 Elem.	55.523	30.052	46.008	81.129

Quick Sort, unsurprisingly, is the fastest method and is particularly efficient when it comes to sorting big amounts of data. For all input sizes, Radix Sort consistently has the longest execution time of the four algorithms. This is perhaps due to its linear temporal complexity. Merge Sort and Heap Sort exhibit comparable performance characteristics, with execution times lying between Quick Sort and Radix Sort.



This graph depicts Quick Sort's efficiency.

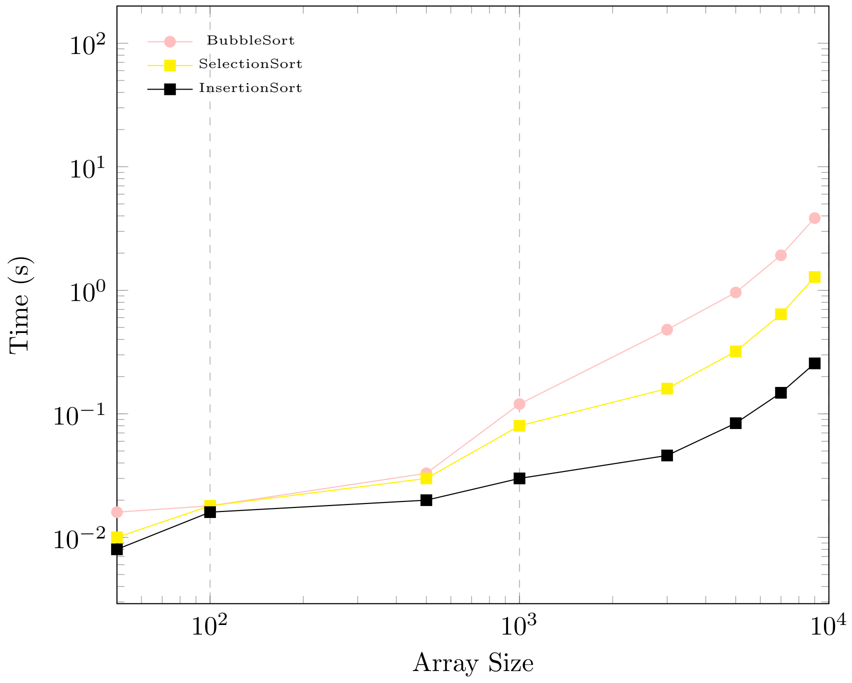
4.2 Performance Comparison of Quadratic Algorithms

As in the previous table, the values from the input file were numbers in random order from 0 to 1000. This table also shows the average time of 4 consecutive tests.

	Bubble Sort	**Insertion Sort**	**Selection Sort**
10 Elem.	0.016	0.009	0.011
50 Elem.	0.024	0.012	0.017
100 Elem.	0.025	0.021	0.023
500 Elem.	0.027	0.026	0.027
1000 Elem.	0.048	0.033	0.046
3000 Elem.	0.09	0.061	0.065
10.000 Elem.	0.75	0.230	0.392
100.000 Elem.	61.63	18.45	33.41
200.000 Elem.	126.77	68.45	86.23

Table 2: Performance for Bubble, Insertion and Selection Sort

When we look at bigger data sets, we can clearly see how inefficient Bubble Sort is when compared to Selection and Insertion Sort. Bubble Sort appears to have a highly consistent run time when the algorithm is performed on smaller data sets, while the other two do not. This might be due to the fact that the overall execution time for tiny data sets is very short, making the differences in efficiency across sorting algorithms less obvious. Only by staring at the bottom half of the table can the entire picture be seen. Insertion Sort is clearly the quickest method for both small and big data sets.



5 Related Work

This section gives an overview of the current research and literature on sorting algorithm comparison that was cited or studied when writing this study.

After consulting [Dolas and Nikam(2012)] paper, it helped compare the performance of several sorting algorithms on both integer and string datasets, and provide an in-depth analysis of the results.

The paper [Karunanithi et al.(2014)] has been of great help in writing this paper. My scripts were modified from this work, which also served as motivation for laying out the benefits and drawbacks of the sorting methods.

The paper [Kumar and Sharma(2013)] compares the performance of several sorting algorithms, including bubble sort, insertion sort, selection sort, quicksort, heapsort, and mergesort, on large data sets, and provides an evaluation of their time complexity and efficiency, providing a great help in writing this paper.

5.1 Is the problem we address new?

Comparing sorting algorithms has long been a well-studied issue in computer science. It is a fundamental problem in algorithm analysis and design that has been extensively explored and evaluated in a variety of settings, including performance analysis, time complexity analysis, and space complexity analysis. However, determining which algorithm is the greatest match for a certain circumstance remains a difficult issue. Each algorithm has its own set of strengths and limitations, thus it's critical to grasp their properties, which is what this study aims to achieve. While this is not a new problem, there is still a need for comprehensive and up-to-date comparisons.However, research in this field is constantly continuing, since new algorithms and enhancements to old algorithms are always being created and assessed.

5.2 What are the advantages/disadvantages?

This study provides a detailed examination of several sorting algorithms, taking into consideration both their theoretical properties and their actual performance on various datasets. Such an examination offers for a better understanding of how the algorithms function and allows for more informed selections when selecting the best algorithm.

Considering that, here are the Advantages and Disadvantages of my paper:

5.2.1 Advantages

1. Provides a thorough evaluation: My research provides a comprehensive review of numerous sorting algorithms, taking into account both theoretical aspects and actual performance on diverse data sets. This detailed examination allows for a more in-depth knowledge of the algorithms' behavior and educated algorithm selection.

2. By evaluating the performance of different sorting algorithms, my paper can provide insights into which algorithms are best suited for certain types of data sets and application scenarios. This information is useful for developers and data scientists who need to select the appropriate algorithm for their unique requirements.

3. All of the materials I utilized to create the experiments are linked and freely accessible. As a result, the findings' applicability is increased.

5.2.2 Disadvantages

1. The evaluation focuses mostly on running time as the primary performance metric. Other critical elements such as space complexity and stability must also be assessed. Some papers do take this into consideration.

2. The quality of my implementation of the sorting algorithms may affect their performance, and differences in implementation may impact the results of the comparison.

3. While the research includes various commonly used sorting algorithms,It does not present all available algorithms,or newly created ones.

6 Conclusions and Future Work

The work provided here aims to address the challenge of picking the best sorting algorithm for a particular circumstance. The paper was intended to have an appealing and simple style so that readers could readily access all of the information they needed. After studying them all, the conclusion is that Insertion Sort is suggested for its simplicity and performance in comparison to the other algorithms tested. Quick Sort proved to be the best when it came to sorting enormous volumes of numbers. The study effectively analyzed numerous sorting algorithms while taking into account varied input data sizes.

It also gave a greater knowledge of the algorithms' behavior and performance through the use of tables and graphs that help visualize the problem.

In summary, the purpose of this work is to describe the most generally used sorting methods in an engaging and appealing manner, making it easier for computer science novices and students to comprehend the fundamentals of sorting.

6.1 Conclusions

Ranking: Sorting Small Amounts of Data

1. Quick Sort	2. Heap Sort	3. Insertion Sort	4. Merge Sort	5. Selection Sort	6. Bubble Sort	7. Radix Sort

Ranking: Sorting Large Amounts of Data

1. Quick Sort	2. Heap Sort	3. Merge Sort	4. Insertion Sort	5. Radix Sort	6. Selection Sort	7. Bubble Sort

References

[Dolas and Nikam(2012)] B. D. Dolas and V. B. Nikam. A comparative study of sorting algorithms on integer and string data sets. *International Journal of Computer Applications*, 50(2):7–14, 2012.

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

[Kumar and Sharma(2013)] A. Kumar and R. Sharma. Sorting algorithms: A comparative analysis. *International Journal of Engineering and Advanced Technology (IJEAT)*, 2:63–67, 2013.