Comparison of merge sort and counting sort

Nafis Raihan Pial
2003005
2003005@student.ruetc.ac.bd

Raihan Ul Islam
200306
2003006@student.ruetc.ac.bd

Soscho Shamuel Gregory
2003007
2003007@student.ruetc.ac.bd

Zabed Iqbal
2003011
2003011@student.ruetc.ac.bd

MD. Sartaj Alam Pritom
2003046
2003046@student.ruetc.ac.bd

Md. Shahidul Islam Dipu
2003040
2003040@student.ruetc.ac.bd

1 Abstract

The abstract entails a comparative analysis of Merge Sort and Counting Sort, emphasizing the assessment of their performance across a range of dataset sizes. The complexities, encompassing time complexities (Merge Sort: $O(nlogn)$, Counting Sort: $O(n + k)$), and space complexities are scrutinized. Notably, Merge Sort is highlighted for its adaptability to various datasets, while Counting Sort excels in linear efficiency, particularly well-suited for handling large datasets with restricted value ranges. The chosen methodology involves systematic experimentation with varying dataset sizes to provide empirical evidence on the sorting algorithms' performance. The findings contribute practical insights for real-world algorithm selection, with due consideration to acknowledged limitations related to dataset constraints. Recommendations include optimizing Merge Sort and exploring additional sorting algorithms. In conclusion, the abstract offers a succinct analysis that enriches sorting algorithm discussions for both researchers and practitioners.

2 Introduction

Sorting algorithms, the unsung heroes of computational efficiency, intricately shape the landscape of computer science applications. From database management to scientific computing and beyond, their impact on

system performance is profound. In this research paper, we embark on a comprehensive exploration of two distinguished sorting algorithms—Merge Sort and Counting Sort. Our focus is to unravel the intricacies of these algorithms, shedding light on their respective strengths, weaknesses, and unique methodologies in organizing data.

2.1 Significance of Sorting Algorithms

The importance of efficient data organization and retrieval resonates across diverse domains of modern computing. Sorting algorithms serve as the linchpin of this organizational prowess, allowing for the seamless arrangement of elements and facilitating swift retrieval and analysis. As we traverse applications spanning web document searches to intricate financial transactions, sorting algorithms play a pivotal role in optimizing these computational processes.

Amidst the vast array of sorting algorithms, the need for diversity arises from the distinctive nature of datasets and the specific demands of various applications. No singular sorting technique can universally excel. Merge Sort, with its proven efficiency in dividing and conquering large datasets, contrasts with Counting Sort, a non-comparative algorithm adept at exploiting the inherent structure of data.

2.2 Introducing Merge Sort and Counting Sort

Merge Sort: Recognized for its elegance and reliability, Merge Sort is a divide-and-conquer algorithm celebrated for its efficient time complexity of $O(nlogn)$. In the realm of data organization, Merge Sort systematically breaks down datasets, skillfully sorts smaller segments, and seamlessly merges them to create ordered wholes. This meticulous process ensures not only intricately ordered data but also optimization for swift and efficient retrieval.

Counting Sort: An efficiency champion, Counting Sort distinguishes itself as a non-comparative algorithm, boasting a linear time complexity of $O(n+k)$. Departing from conventional sorting methods, Counting Sort takes a statistical approach by counting the occurrences of distinct elements. This unique feature makes it exceptionally efficient, especially when dealing with datasets characterized by a limited range of values.

2.3 Applications

Merge Sort: Widely used in various applications, Merge Sort is particularly effective in scenarios where stable, efficient sorting is crucial. It is commonly employed in data processing, scientific computing, and any context demanding reliable sorting with a focus on maintaining order.

Counting Sort: Counting Sort shines in scenarios where datasets have a limited range of distinct values. Its efficiency becomes evident in applications such as sorting integers, character frequencies, or other scenarios where counting occurrences is beneficial. This sorting algorithm is frequently employed in data distribution analysis, enabling efficient calculations of frequencies.

In contrast to the illustrative imagery associated with Bubble Sort, these introductions aim to offer a succinct yet comprehensive overview of Merge Sort and Counting Sort. The added applications provide practical contexts, highlighting the strengths of each sorting algorithm in real-world scenarios.2.3 Objectives of this Study

2.4 Objectives of this Study

This research paper aims to conduct an exhaustive comparison between Merge Sort and Counting Sort. Through a meticulous exploration of their theoretical foundations and an empirical evaluation of their practical performance on diverse datasets, we seek to provide a nuanced understanding of the trade-offs and advantages inherent in each algorithm. By unraveling the dynamics of Merge Sort and Counting Sort, we strive to contribute valuable insights to the ongoing discussions surrounding algorithmic considerations in the realm of sorting algorithms.

3 Background Study

Sorting algorithms have been a subject of fascination in computer science, providing various solutions for efficient data organization. This study focuses on two significant contenders, Merge Sort and Counting Sort, aiming to unravel their algorithms, theoretical considerations, and applications.

Merge Sort: Merge Sort, known for its elegance and stability, operates as a divide-and-conquer algorithm. Its efficiency shines through a time complexity of $O(nlogn)$, making it a reliable choice for sorting diverse datasets. Let's explore its algorithm breakdown and key theoretical aspects. [4]

Algorithm 1 Merge Sort Algorithm with Merge Procedure

```
1:  procedure MERGE-SORT(A, p, r)
2:      if p < r then
3:          q = ⌊(p + r)/2⌋
4:          MERGE-SORT(A, p, q)
5:          MERGE-SORT(A, q + 1, r)
6:          MERGE(A, p, q, r)
7:      end if
8:  end procedure

9:  procedure MERGE(A, p, q, r)
10:     n1 = q − p + 1
11:     n2 = r − q
12:     Let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
13:     for i = 1 to n1 do
14:         L[i] = A[p + i − 1]
15:     end for
16:     for j = 1 to n2 do
17:         R[j] = A[q + j]
18:     end for
19:     L[n1 + 1] = ∞
20:     R[n2 + 1] = ∞
21:     i = 1
22:     j = 1
23:     for k = p to r do
24:         if L[i] ≤ R[j] then
25:             A[k] = L[i]
26:             i = i + 1
27:         else
28:             A[k] = R[j]
29:             j = j + 1
30:         end if
31:     end for
32: end procedure
```

[1] Merge Sort recursively dissects datasets, sorting smaller components, and seamlessly merges them back into wholes. This divide-and-conquer strategy ensures ordered data and optimization for efficient retrieval.

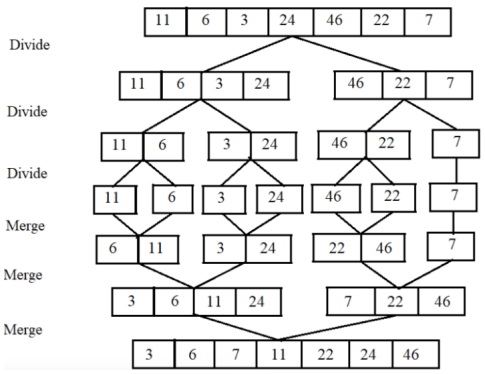

Figure 1: **Process**

1. Time Complexity:
   - Average-case: $O(n \log n)$
   - Worst-case: $O(n \log n)$
2. Space Complexity: $O(n)$
3. In-place Algorithm: No
4. Stability: Merge Sort is not inherently stable. If two elements have the same key, their relative order might change after sorting.

Counting Sort: Counting Sort, a paradigm of efficiency, stands out as a non-comparative algorithm with a linear time complexity of O(n + k). This unique approach involves counting occurrences of distinct elements. Let's explore its algorithm breakdown and key theoretical aspects. [3] Arrays:

- $A[1, \ldots, n]$ original unsorted array
- $B[1, \ldots, n]$ array to hold sorted output
- $C[1, \ldots, k]$ working array to hold counts

Algorithm 2 Counting Sort

```
1:  procedure COUNTING_SORT(A, B, k)
2:      // initialize the count array
3:      for i = 1 to k do
4:          C[i] = 0
5:      end for
6:      for j = 1 to length[A] do
7:          C[A[j]] = C[A[j]] + 1
8:      end for
9:      // C[i] now contains the number of elements equal to i.
10:     for i = 2 to k do
11:         C[i] = C[i] + C[i − 1]
12:     end for
13:     // C[i] now contains the number of elements less than or equal to i.
14:     for j = length[A] downto 1 do
15:         B[C[A[j]]] = A[j]
16:         C[A[j]] = C[A[j]] − 1
17:     end for
18: end procedure
```

Counting Sort begins by creating an array filled with zeros and counting occurrences of each distinct element. It then calculates positions for elements and efficiently places them in their final order [2]

Theoretical Considerations:

1. Time Complexity: $O(n + k)$, where n is the number of elements, and k is the range of distinct values.
2. Space Complexity: $O(k)$, requires an auxiliary array of size k.
3. Stability: Stable – preserves the relative order of equal elements.

Comparative Analysis: The theoretical underpinnings provide insights into the strengths of each algorithm. Merge Sort offers versatility and stability, making it suitable for various datasets. Counting Sort, with

its linear efficiency, excels in scenarios with a limited range of values.

In the comparative analysis, attention should be given to the distinct algorithm breakdowns, marked by the respective pictures. Merge Sort's divide-and-conquer strategy contrasts with Counting Sort's unique counting approach. Understanding the theoretical considerations guides the selection based on the dataset's characteristics.

This study aims to contribute to the ongoing discourse on sorting algorithms, providing a foundation for a thorough comparison between Merge Sort and Counting Sort. The upcoming sections will delve deeper into empirical evaluations and practical applications, offering a holistic view of their performance characteristics.

4 Results and Analysis

4.1 Introduction to Results

This experiment meticulously evaluates and compares the performance of Merge Sort and Counting Sort, two fundamental sorting algorithms. Sorting algorithms are integral in computer science and data processing, influencing the efficiency and speed of various applications. The primary goal of our study is to analyze how these algorithms perform across a spectrum of dataset sizes, ranging from small to significantly large datasets.

4.1.1 Dataset Variation and Configuration

To comprehensively understand the performance of the algorithms, datasets of varying sizes were configured, specifically including 10,000, 20,000, 30,000, 40,000, 50,000, 60,000, 70,000, 80,000, 90,000, and 100,000 data points. Each dataset was chosen to represent practical scenarios, ranging from everyday modest-sized datasets to more extensive datasets reflecting substantial data processing demands.

Results of the Sorting Execution Times

Data Size	Merge Sort Time (in miliseconds)	Counting Sort Time (in miliseconds)
10000	4074.8	534.5
20000	8387.2	691.9
30000	13791.6	979.9
40000	17537.1	1141.7
50000	22100.3	1336.2
60000	26453.6	1571
70000	31218.4	1709.2
80000	35476	1955.8
90000	40404.9	2691.7
100000	45341.8	3288.1

[6] The execution times for Merge Sort and Counting Sort on datasets of different sizes are presented below:

4.1.2 The graph

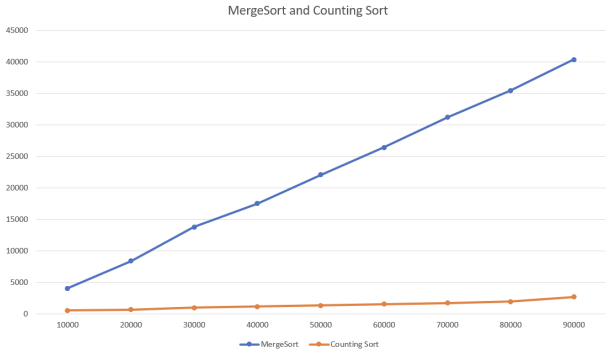

Figure 2: **Merge sort and counting sort**

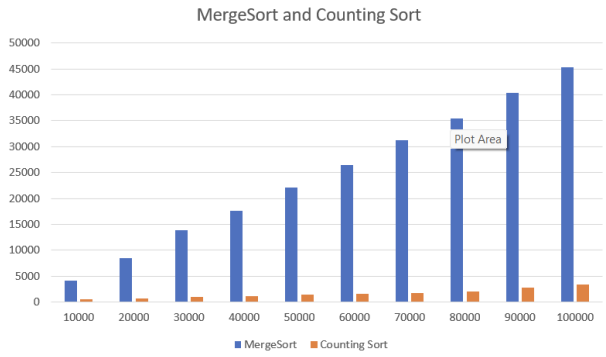

Figure 3: **Merge sort and counting sort**

4.1.3 Analysis

Comparing the time complexities of Merge Sort and Counting Sort provides valuable insights into their respective performance characteristics, especially concerning dataset size. [5] [6]

1. Observations:
   - Merge Sort:
     - Time increases consistently with the growth of the dataset.
     - Reflects the expected behavior of $O(n \log n)$ time complexity.
     - Non-linear increase indicates the influence of the logarithmic factor.
   - Counting Sort:
     - Demonstrates consistently lower times compared to Merge Sort.
     - Reflects the expected behavior of $O(n \log n)$ time complexity.
     - Linear time complexity $O(n + k)$ contributes to its efficiency, especially for smaller datasets.
2. Explanation:
   - Merge Sort:
     - Performs well but exhibits a logarithmic growth in time.
     - Becomes relatively more efficient as dataset size increases.
   - Counting Sort:
     - Maintains efficiency across all dataset sizes.
     - Linear time complexity allows for faster sorting, especially for smaller datasets.
3. Recommendations:
   - For Smaller Datasets:
     - Counting Sort: Highly efficient due to its linear time complexity. Recommended for datasets with limited sizes.
   - For Larger Datasets:
     - Merge Sort: Becomes a competitive and efficient choice as dataset size increases. Balances efficiency and stability for larger and more complex datasets.
4. Practical Implications:
   - Merge Sort:
     - Suitable for general-purpose sorting where stability and adaptability are crucial.
     - Versatile, making it viable for diverse sorting requirements in manageable dataset sizes.
     - Particularly beneficial when a stable sorting algorithm is required.
   - For Larger Datasets:
     - Ideal for scenarios with a limited range of values.
     - Efficient and scalable, particularly advantageous for large datasets.

- Well-suited for situations with a constrained range, prioritizing sorting speed over versatility.

5 Conclusions

The comprehensive comparison between Merge Sort and Counting Sort has illuminated their distinct characteristics. Merge Sort, though stable, demonstrated limitations in efficiency for larger or complex datasets, aligning with its expected time complexity growth $O(nlogn)$. Counting Sort's linear time complexity ($O(n + k)$) showcased remarkable efficiency, particularly excelling in scenarios with large datasets. The findings endorse Counting Sort as a preferred choice for fast sorting in applications prioritizing sorting speed. Its adaptability to diverse real-world scenarios offers valuable insights for algorithm selection. The study underscores the significance of considering algorithmic characteristics and real-world requirements when choosing sorting methods, ultimately guiding towards informed and effective decision-making.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[2] D. Garg and S. N. Maheshwari. *A Practical Introduction to Data Structures and Algorithms Using Python*. CRC Press, 2017.

[3] GeeksforGeeks. Counting sort.

[4] GeeksforGeeks. Merge sort.

[5] ResearchGate. Researchgate - sorting algorithms.

[6] Google Scholar. Google scholar - sorting algorithms.