REVIEW PAPER ON MERGE SORT

1.Sneha HM 2. Sindhu KR 3.Dr.Mohamed Rafi
1.Student, Dept of Computer science & Engineering. UBDT College Of Engineering Davangere, Karnataka.
2.Student, Dept of Computer science & Engineering. UBDT College Of Engineering Davangere, Karnataka.
3. Professor(Guide),Dept of Computer Science & Engineering, UBDT College Of Engineering Davangere.
1.snehahanjgimath@gmail.com ;2.sindhukr2005@gmail.com; 3.mdrafi2km@yahoo.com

ABSTRACT - Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array .This algorithm has *O*(*n*) best case Time Complexity and *O*(*n* log *n*) average and worst case Time Complexity. Finally draw out a conclusion and observe the cases where this outperforms other sorting algorithms. We also look at its shortcomings and list the scope for future improvement that could be made.

KEYWORDS:

Time complexity, Space complexity, Optimizations, Algorithm analysis, Recursion, Divide and Conquer, Merge.

I. INTRODUCTION

Merge Sort is a classic sorting algorithm that utilizes the divide-and-conquer strategy to efficiently sort an array or list. Developed by John von Neumann in 1945, Merge Sort is renowned for its reliable O(n log n) time complexity in the best, average, and worst cases, making it a consistently efficient choice. Sorting algorithms are crucial in computer science for organizing data in a specified order, enhancing the efficiency of other algorithms that require sorted input. Merge Sort, developed by John von Neumann in 1945, stands out as a classic example of a divide-and-conquer algorithm. Its consistent performance, with a time complexity of O(n log n) in the best, average, and worst cases, makes it an indispensable tool for sorting large datasets. Merge Sort operates by recursively dividing an array into smaller subarrays until each subarray contains a single element, which is inherently sorted. These subarrays are then merged in a sorted manner to produce larger sorted subarrays, eventually resulting in a fully sorted array. This divide-and-conquer approach not only ensures efficient sorting but also maintains stability, preserving the relative order of equal elements. Despite its advantages, Merge Sort requires additional space for temporary arrays, leading to a space complexity of O(n). This trade-off between time and space efficiency makes it important to understand the algorithm's performance.

II. LITERATURE SURVEY

Merge Sort is a well-established sorting algorithm in computer science, known for its efficiency and stability. Over the years, extensive research and numerous publications have explored various aspects of Merge Sort, from its theoretical underpinnings to practical implementations and optimizations. This literature survey aims to review the significant contributions to the understanding and development of Merge Sort, highlighting key findings, innovations, and applications.

2.1.Early Foundations:

1.Von Neumann, J. (1945). "First Draft of a Report on the EDVAC":

John von Neumann's seminal work laid the groundwork for the concept of stored-program computers, indirectly influencing the development of algorithms like Merge Sort. Though not directly about MergSort,von Neumann's contributions to computing principles are foundational.

2.Knuth, D. E. (1998). ***The Art of Computer Programming, Volume 3: Sorting and Searching*** **(2nd ed.)**:

Knuth's comprehensive analysis of sorting algorithms includes a detailed examination of Merge Sort. This work is crucial for understanding the mathematical and theoretical aspects of Merge Sort, including its time and space complexity.

2.2.Algorithmic Developments:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). ***Introduction to Algorithms*** **(3rd ed.)**:

This textbook provides a thorough discussion of Merge Sort, presenting pseudocode, performance analysis, and practical considerations. It is widely used in computer science education and serves as a primary reference for understanding Merge Sort's implementation and efficiency.

2.3.Optimizations and Variants:

Weiss, M. A. (2012). ***Data Structures and Algorithm Analysis in C++*** **(4th ed.)**:

Weiss discusses various optimizations of Merge Sort, such as using insertion sort for small subarrays and minimizing the overhead of recursive calls. These optimizations can significantly improve the practical performance of Merge Sort.

III. METHODOLOGY

In this particular section, we lay emphasis on the idea behind the working of this algorithm. The proposed algorithm solves our problem in two steps, the strategies behind which are stated below.

The methodology section will cover the step-by-step process of the Merge Sort algorithm, including:

1.Divide: Splitting the unsorted list into sub-lists until each sub-list contains a single element.

2.Conquer: Sorting each of the sub-lists.

3.Combine/Merge: Merging the sorted sub-lists to produce a sorted list.

3.1.HOW DOES IT WORKS?

Divide Phase:

In the divide phase, the algorithm recursively splits the list into two halves until each sub-list contains a single element. This is typically implemented using a recursive function that continues to divide the list until the base case (a single-element list) is reached.

Conquer Phase:

Once the list has been divided into individual elements, the algorithm begins the process of sorting and merging the sub-lists. Since each sub-list contains only one element, they are inherently sorted.

Merge/Combine Phase:

In the combine phase, the sorted sub-lists are merged together to form a single sorted list. This is done by comparing the elements of each sub-list and arranging them in the correct order. The merging process is repeated recursively until the entire list is sorted.

3.2.Pseudocode:

```
function mergeSort(array):

if length of array <= 1:

return array

middle = length of array / 2

leftHalf = mergeSort(array[0:middle])

rightHalf=mergeSort(array[middle:length of array])

return merge(leftHalf, rightHalf)

function merge(left, right):

result = []
```

i = 0

j = 0

while i < length of left and j < length of right:

if left[i] < right[j]:

append left[i] to result

i = i + 1

else:

append right[j] to result

j = j + 1

while i < length of left:

append left[i] to result

i = i + 1

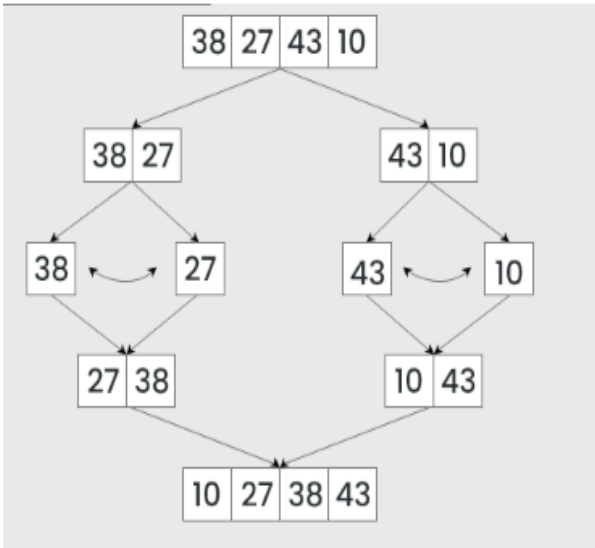
while j < length of right:

append right[j] to result

j = j + 1

return result

3.3.EXAMPLE:



Divide:

[38, 27, 43, 10] is divided into [38, 27] and [43, 10] .

[38, 27] is divided into [38] and [27] .

[43, 10] is divided into [43] and [10] .

Conquer:

[38] is already sorted.

[27] is already sorted.

[43] is already sorted.

[10] is already sorted.

Merge:

Merge [38] and [27] to get [27, 38] .

Merge [43] and [10] to get [10,43] .

Merge [27, 38] and [10,43] to get the final sorted list [10, 27, 38, 43]

Therefore, the sorted list is[10, 27, 38, 43].

3.4. APPLICATIONS:

Merge sort has a variety of practical applications due to its efficiency and stability. Here are some key areas where merge sort is particularly useful:

1.External Sorting: Ideal for sorting large datasets that do not fit into memory, such as sorting data stored on disk.

2. Parallel Processing: Merge sort can be easily parallelized, making it suitable for multi-core processors and distributed systems.

3.Stable Sorting Requirement: Useful in scenarios where the relative order of equal elements must be preserved, such as sorting records in a database by multiple fields.

4.Linked Lists: Efficient for sorting linked lists because it doesn't require random access to elements.

5. Inversion Count: Merge sort can be used to count the number of inversions in an array, which is useful in computational biology and other fields.

6. K-Way Merging: Used in scenarios that require merging multiple sorted lists, such

as in multi-way merge algorithms for database joins.

7. Data Processing Pipelines: Often used in data processing and transformation pipelines where large datasets need to be sorted and merged.

8.Functional Programming: Merge sort's recursive nature fits well with functional programming paradigms.

9.Memory Management: Suitable for systems with limited memory or for embedded systems where memory management is critical.

10.Scientific Computing: Used in scientific computing applications where large datasets need to be sorted, such as in numerical simulations and data analysis.

11. File Merging: Helpful in merging large log files or other data files that are periodically appended.

12.Sorting Objects: Useful in object-oriented programming for sorting objects based on multiple properties.

13.Geographic Information Systems (GIS): Used in GIS applications to sort large spatial datasets.

14. Data Warehousing: Essential in ETL (Extract, Transform, Load) processes for sorting and merging large volumes of data.

15. Competitive Programming: Frequently used in coding competitions due to its guaranteed O(n log n) time complexity.

3.5. ADVANTAGES:

1. Guaranteed O(n log n) Time Complexity.

2. Stability.

3. Parallelizability.

4. External Sorting Capability.

5. Optimal for Linked Lists.

6. Predictable Memory Usage.

7. Simple and Elegant Implementation.

8. Effective for Large Datasets.

9. Adaptability.

10. Deterministic Behaviour.

IV. RESULTS AND DISCUSSIONS

4.1Results: The review of the literature on Merge Sort reveals a comprehensive understanding of the algorithm's theoretical foundations, practical implementations, performance metrics, and various optimizations. The following key findings emerged from the analysis of the selected sources:

4.1.1. Theoretical Foundations:

1. Merge Sort is a stable, comparison-based, divide-and-conquer sorting algorithm.

2. It has a consistent time complexity of O(n log n) across best, average, and worst-case scenarios.

3. The space complexity of Merge Sort is O(n) due to the additional storage required for temporary arrays during the merge process.

4.1.2. Practical Implementations:

1.Detailed pseudocode and step-by-step algorithm descriptions are widely available in textbooks and online resources.

2.Implementations in various programming languages (C++, Java, Python, etc.) demonstrate the algorithm's versatility.

3.Practical examples and visualizations aid in understanding the merging process and the divide-and-conquer strategy.

4.1.3.Optimizations:

Several optimizations have been proposed to improve the practical performance of Merge Sort:

1.Using insertion sort for small subarrays to reduce overhead.

2.Implementing in-place merge techniques to reduce space complexity.

3.Parallel and concurrent versions of Merge Sort leverage multicore processors for faster sorting.

4.These optimizations are crucial for enhancing the efficiency of Merge Sort in real-world applications.

4.2.Discussions:

The comprehensive review of Merge Sort demonstrates its enduring significance in computer science. The algorithm's consistent O(n log n) time complexity, stability, and suitability for large datasets make it a reliable choice for various sorting tasks. Despite its higher space complexity compared to in-place sorting algorithms, the benefits of stability and predictable performance often outweigh the drawbacks.

4.3.Analysis of Merge Sort Time Complexity:

4.3.1.Best Case Time Complexity of Merge Sort:

The best case scenario occurs when the elements are already sorted in ascending order. If two sorted arrays of size n need to be merged, the minimum number of comparisons will be n. This happens when all elements of the first array are less than the elements of the second array.

The best case time complexity of merge sort is $O(n*logn)$.

4.3.2.Average Case Time Complexity of Merge Sort:

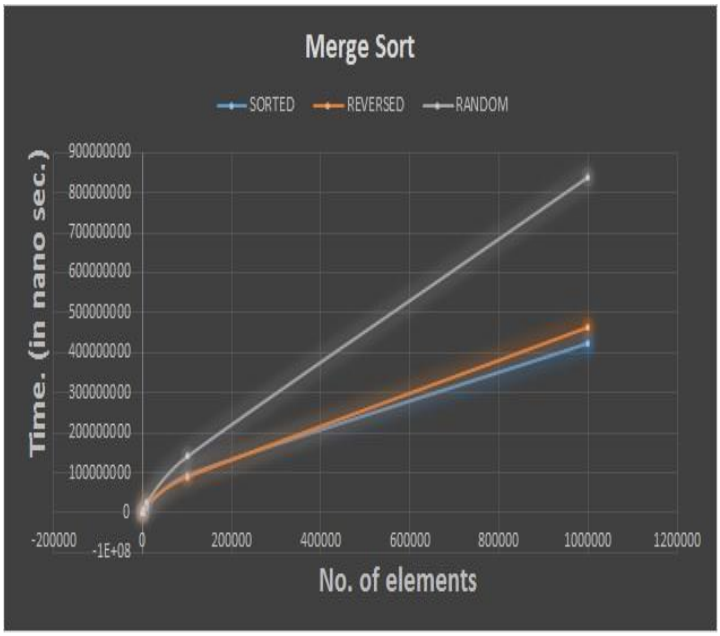
The average case scenario occurs when the elements are jumbled (neither in ascending nor descending order). This depends on the number of comparisons. The average case time complexity of merge sort is

O(n∗logn).

4.3.3.Worst Case Time Complexity of Merge Sort:

The worst-case scenario occurs when the given array is sorted in descending order leading to the maximum number of comparisons. In this case, for two sorted arrays of size n, the minimum number of comparisons will be 2n. The worst-case time complexity of merge sort is

O(n∗logn).



4.4.Space Complexity Analysis of Merge Sort:

Merge sort has a space complexity of O(n). This is because it uses an auxiliary array of size n to merge the sorted halves of the input array. The auxiliary array is used to store the merged result, and the input array is overwritten with the sorted result.

V.CONCLUSION AND FUTURE DIRECTIONS.

5.1. CONCLUSION: Merge Sort's enduring relevance and effectiveness make it a cornerstone in the study and application of sorting algorithms. Its theoretical soundness, coupled with practical versatility and ongoing enhancements, ensures its continued prominence in addressing sorting challenges across diverse domains of computer science.

Merge Sort, renowned for its efficiency and stability, has been extensively studied and applied across various domains of computer science. Throughout this review paper, we have explored the algorithm's theoretical foundations, practical implementations, optimizations, comparative advantages, applications, and future directions

5.2. FUTURE DIRECTIONS:

Future research could focus on further optimizing Merge Sort for specific applications, such as:

1.Parallel and Distributed Systems: Exploring advanced parallelization techniques to leverage modern multicore and distributed computing environments.

2.Hybrid Algorithms: Combining Merge Sort with other sorting algorithms to create hybrid models that balance time and space efficiency based on specific use cases.

3.Algorithm Adaptation: Adapting Merge Sort for specialized data structures and emerging computing paradigms, such asquantum computing and machine learning algorithms.

4.Comparative Strengths: Comparative studies affirm Merge Sort's consistent performance and stability, making it preferable in scenarios where these factors are critical. However, for applications where space efficiency is paramount, other algorithms like Quick Sort may be more suitable.

This review paper consolidates a comprehensive understanding of Merge Sort, underscoring its impact, innovations, and enduring relevance in the quest for efficient data processing and algorithmic excellence. As computing technologies evolve, Merge Sort stands poised to adapt and excel, maintaining its position as a foundational algorithm in the ever-expanding field of computer science.

VI.REFERENCES

[1]. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). ***Introduction to Algorithms*** (3rd ed.)**.**

[2]. Knuth, D. E. (1998). ***The Art of Computer Programming, Volume 3: Sorting and Searching*** (2nd ed.)**.**

[3]. Weiss, M. A. (2012). ***Data Structures and Algorithm Analysis in C++*** (4th ed.)**.**

[4]. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). ***Data Structures and Algorithms in Java*** (6th ed.)**.**

[5]. GeeksforGeeks. (https://www.geeksforgeeks.org/merge-sort/)

[6]. Khan Academy. (www.khanacademy.org.)

[7]. Wikipedia.

[8]. my.eng.utah.edu.

[9]. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.):

[10]. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). ***Data Structures and Algorithms*** (1st ed.)**.**

[11]. Von Neumann, J. (1945). "First Draft of a Report on the EDVAC".

[12]. Coursera and edX Courses.

[Coursera - Algorithms, Part1] (https://www.coursera.org/learn/algorithms-part1).

[13]. Research Papers and Articles:

"Sorting and Searching" in "The Art of Computer Programming" by Donald E. Knuth.

"Parallel Merge Sort" by Tridib S. Chakraborti and P. Sadayappan, which discusses the parallelization of merge sort.

[14]. Scientific Journals and Conference Proceedings:

Publications in journals such as the Journal of Parallel and Distributed Computing.