# A PROJECT REPORT

**ON**

# “Implemetation of Merge sort and multithreded merge sort”

**SUBMITTED BY:**

Bhoite Shubham Rama

**UNDER THE GUIDENCE OF:**

**Prof. Sayyad J. I.**

**DEPARTMENT OF COMPUTER ENGINEERING**

**HSBPVT’s PARIKRAMA COLLEGE OF ENGINEERING, KASHTI**
**(Approved by AICTE & Affiliated to Savitribai Phule Pune university)**

**Year of submission: 2023-2024**

## Abstract

This report delves into the practical implementation and performance evaluation of the merge sort algorithm and its multithreaded counterpart, shedding light on their efficiency and applicability in diverse scenarios. Merge sort is a well-established sorting algorithm renowned for its reliability and O(n log n) time complexity. In contrast, multithreaded merge sort harnesses the power of parallelism to enhance sorting performance. Through rigorous testing, we compare the execution times of both algorithms and analyze their behavior in best and worst-case scenarios. This study provides valuable insights for selecting the most suitable sorting method based on specific requirements and available computational resources.

## Problem Statement

The core challenge addressed in this report pertains to the efficient implementation and comparative analysis of the merge sort algorithm and its multithreaded counterpart. Sorting algorithms are fundamental in computer science, with merge sort standing out as a well-known, efficient, and stable sorting method boasting a time complexity of O(n log n). The problem at hand extends to evaluating multithreaded merge sort, a parallelized approach designed to further enhance sorting efficiency by leveraging multiple processing threads.

In this context, the problem encompasses:

1. Implementing the merge sort algorithm to provide a baseline for comparison.

2. Implementing multithreaded merge sort, leveraging parallelism to expedite sorting tasks.

3. Conducting a comprehensive performance analysis to compare the execution times of both algorithms.

4. Analyzing the performance of merge sort and multithreaded merge sort across a spectrum of scenarios, including best and worst-case situations.

## Motivation

The motivation behind undertaking this study is rooted in the fundamental role of sorting algorithms in the realm of computer science and their pervasive impact on real-world applications. Sorting is a ubiquitous task, essential for organizing data in a structured manner, and it serves as a building block for a myriad of algorithms and software systems. This intrinsic significance drives our exploration of sorting techniques, particularly the merge sort algorithm and its multithreaded counterpart.

Key motivators for this study include:

1. Efficiency in Data Processing: Sorting large datasets efficiently is crucial in various fields, from database management to scientific computing. Optimizing sorting algorithms can lead to significant performance improvements.

2. Time Complexity Considerations: Merge sort has long been celebrated for its O(n log n) time complexity, making it an attractive option for sorting tasks. The motivation here lies in understanding whether the multithreaded approach can further expedite the process.

3. Parallel Computing's Promise: In an era of increasingly parallelized hardware, there is a growing need to harness the power of multithreading and parallel processing. Multithreaded merge sort aligns with this trend and offers the prospect of improved performance.

4. Real-World Relevance: Sorting algorithms play a pivotal role in real-world applications such as search engines, e-commerce, data analytics, and many more. The motivation stems from the practical impact that optimizing sorting methods can have on these domains.

5. Resource Optimization: In scenarios where computational resources are limited, the choice of a sorting algorithm becomes crucial. Understanding which algorithm is best suited for specific resource constraints is a primary driver for this study.

## Objectives

The primary objectives of this study encompass the implementation, comparative analysis, and performance evaluation of the merge sort algorithm and its multithreaded variant. These objectives serve as a framework for achieving a deeper understanding of sorting algorithms and their adaptability to various scenarios:

1. Implementation of Merge Sort: The foremost objective is to implement the merge sort algorithm to establish a baseline for comparison. This involves coding, testing, and verifying the correctness of the merge sort algorithm.

2. Implementation of Multithreaded Merge Sort: To explore the benefits of parallelism, another key objective is to implement the multithreaded merge sort algorithm. This entails developing a multithreaded approach and ensuring that it functions correctly.

3. Performance Comparison: A crucial objective is to conduct a comprehensive performance analysis of both merge sort and multithreaded merge sort. This comparison will involve executing both algorithms on datasets of varying sizes and complexities to assess their efficiency.

4. Time Complexity Analysis: Perform an in-depth time complexity analysis to understand how each algorithm behaves under different conditions. Examine factors affecting their runtime performance.

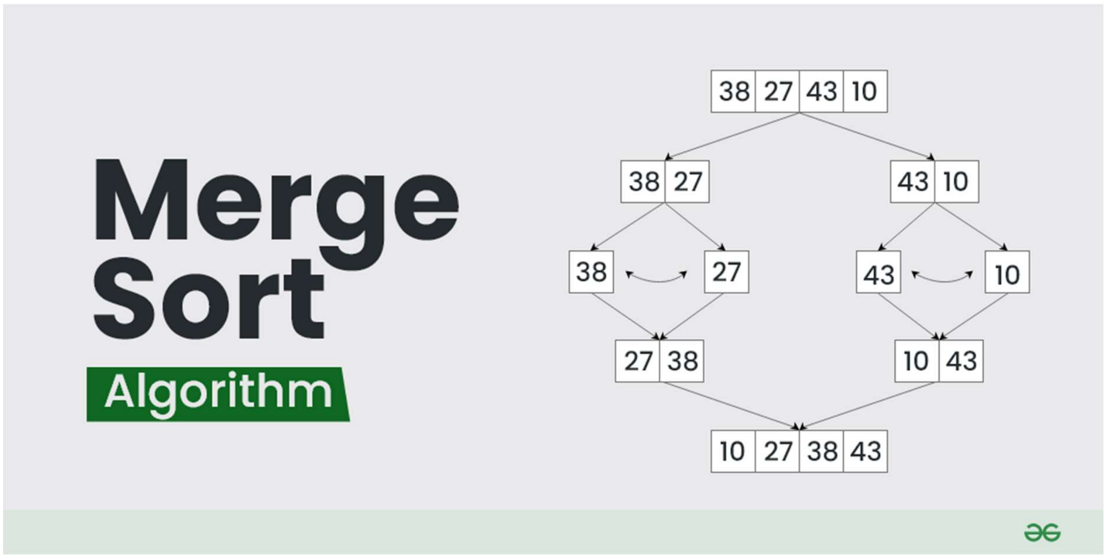
5. Scenario-Based Assessment: Evaluate the performance of merge sort and multithreaded merge sort in various scenarios, including best-case, worst-case, and average-case scenarios. Identify strengths and weaknesses in each context.

6. Resource Utilization: Explore the resource utilization aspect by examining how each algorithm consumes computational resources, including memory and processing power.

7. Recommendations: Based on the findings, formulate recommendations for the selection of the most appropriate sorting algorithm in practical applications. Offer guidance on when to opt for merge sort or multithreaded merge sort based on specific requirements and available resources.

## Merge Sort

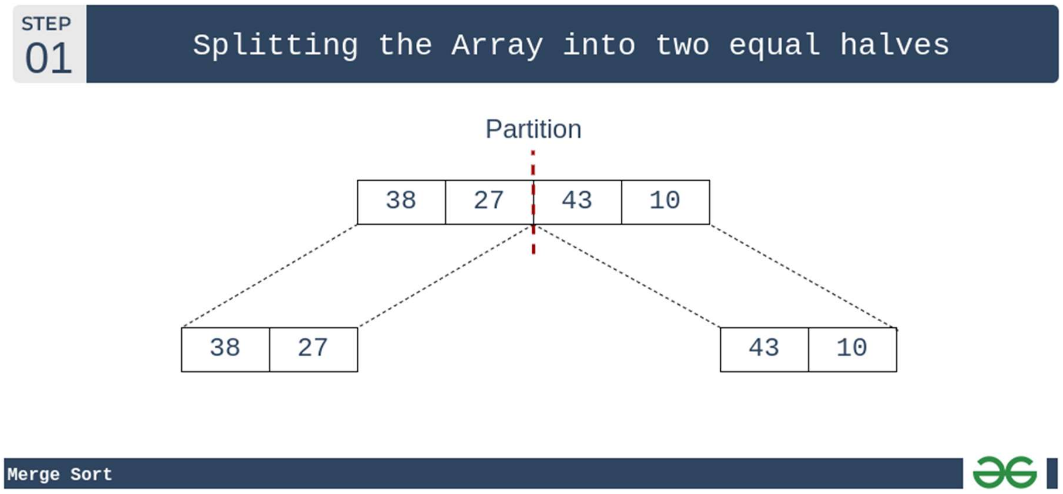
**Merge sort** is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



*Merge Sort Algorithm*

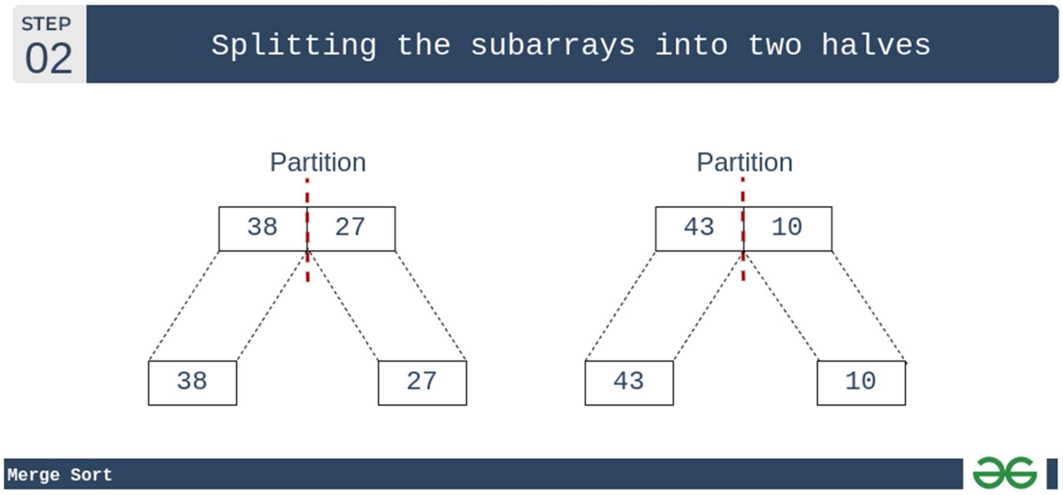
**How does Merge Sort work?**
Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.
See the below illustration to understand the working of merge sort.
**Illustration:**
Lets consider an array **arr[] = {38, 27, 43, 10}**

- Initially divide the array into two equal halves:



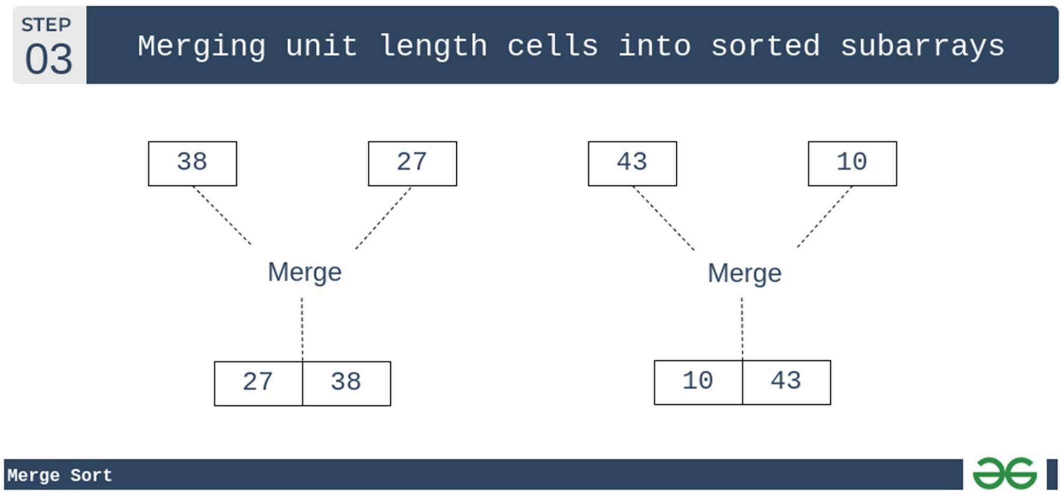
*Merge Sort: Divide the array into two halves*

- These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



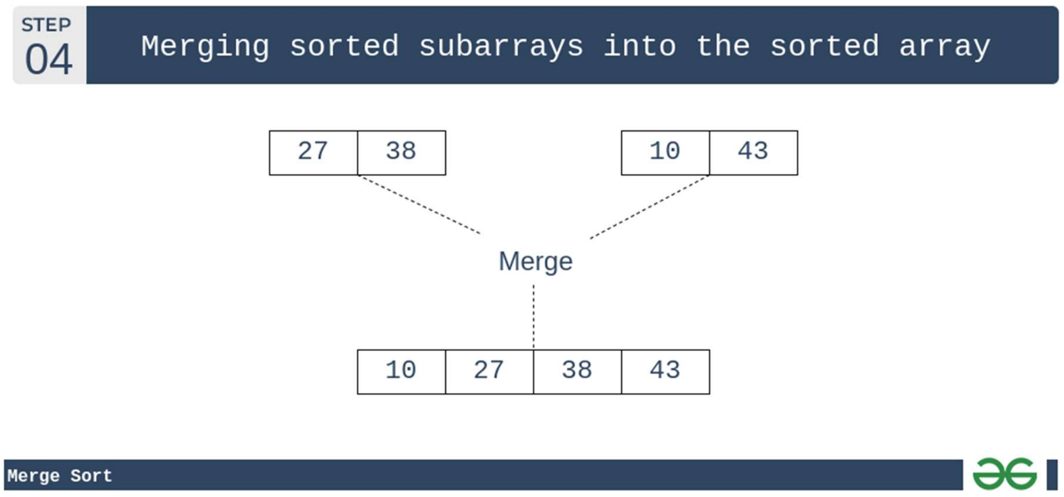
*Merge Sort: Divide the subarrays into two halves (unit length subarrays here)*

These sorted subarrays are merged together, and we get bigger sorted subarrays.



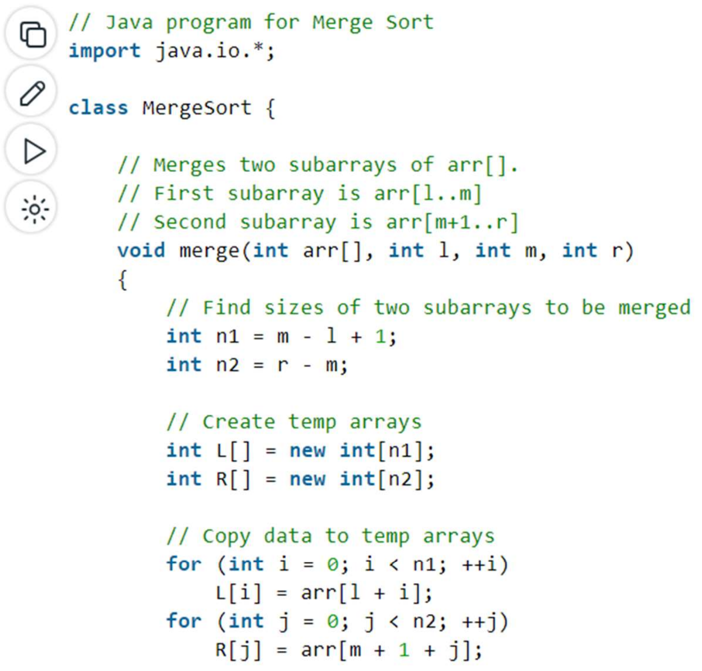
*Merge Sort: Merge the unit length subarrys into sorted subarrays*

This merging process is continued until the sorted array is built from the smaller subarrays.



*Merge Sort: Merge the sorted subarrys to get the sorted array*

## Implementation of code



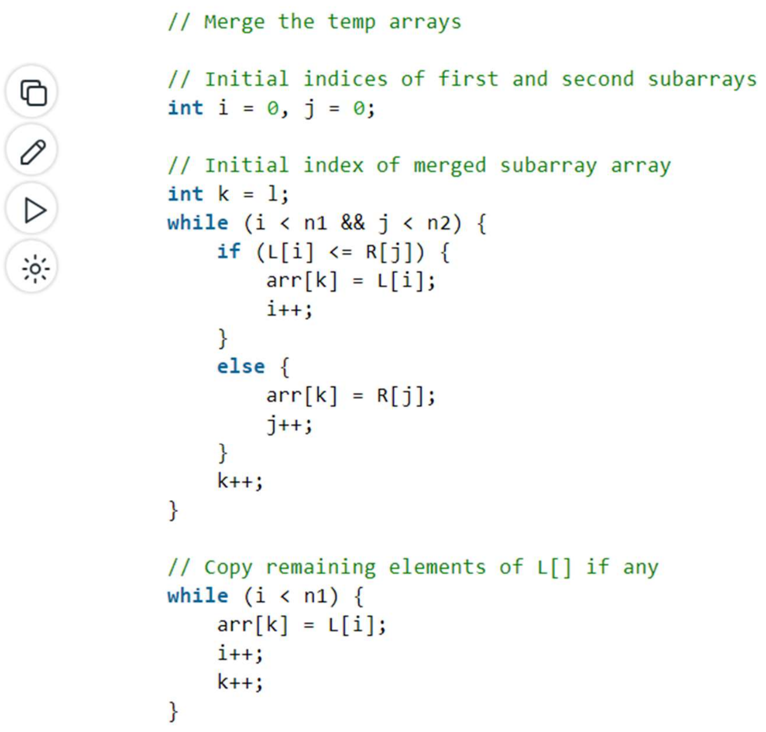
```java
// Java program for Merge Sort
import java.io.*;

class MergeSort {

    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        // Create temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data to temp arrays
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];
```



```java
        // Merge the temp arrays

        // Initial indices of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of L[] if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        // Copy remaining elements of R[] if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int l, int r)
    {
        if (l < r) {

            // Find the middle point
            int m = l + (r - l) / 2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr, m + 1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    // A utility function to print array of size n
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver code
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };

        System.out.println("Given array is");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length - 1);

        System.out.println("\nSorted array is");
        printArray(arr);
    }
}
```

**Output**

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

**Complexity Analysis of Merge Sort:**
**Time Complexity:** O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \theta(n)$
The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
**Auxiliary Space:** O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

## Applications of Merge Sort

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem

**Advantages of Merge Sort:**

- **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

**Drawbacks of Merge Sort:**

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

## Conclusion

This study underscores the enduring efficiency of the merge sort algorithm and the potential of multithreaded merge sort for expedited sorting. The choice between them hinges on specific scenarios and resource availability. Merge sort remains a reliable, resource-friendly option, while multithreaded merge sort shines in resource-rich environments. Our findings provide decision-makers with valuable guidance for selecting the most suitable sorting algorithm based on their needs and resources.