
	<p style="text-align: center;">Pimpri Chinchwad Education Trust's  <b>Pimpri Chinchwad College of Engineering (PCCoE)</b>          (An Autonomous Institute)          Affiliated to Savitribai Phule Pune University (SPPU)          ISO 21001:2018 Certified by TUV</p>	
<b>Department:</b> Information Technology <b>Academic Year:</b> 2025-26 <b>Semester:</b> V		
<b>DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY</b>		

## ASSIGNMENT NO.1

**Aim:-**The aim of this study is to design and implement an efficient sorting algorithm using Merge Sort to arrange customer orders based on their timestamps. The solution should be scalable to large datasets (up to 1 million orders) while minimizing computational overhead and enabling performance comparison with traditional sorting techniques.

### Objective:

- To model the problem of arranging customer orders as a sorting problem based on timestamps.
- To implement the Merge Sort algorithm for efficient large-scale sorting.
- To analyze and compare Merge Sort with traditional sorting algorithms (e.g., Bubble Sort, Insertion Sort, Quick Sort).
- To evaluate performance on large datasets (up to 1 million orders).

### Problem Statement:

Scenario: Customer Order Management

In large-scale e-commerce and retail applications, millions of customer orders are generated daily, each having a timestamp that records when the order was placed. For tasks such as processing, analytics, and delivery scheduling, it is essential to sort these orders chronologically.

Your task as a system designer is:

- To implement Merge Sort to arrange orders by timestamp.
- To ensure the algorithm can handle up to 1 million orders efficiently.
- To analyze the time complexity of Merge Sort and compare it with other sorting algorithms.

### Explanation:

Sorting is a fundamental operation in computer science, crucial for database management, search optimization, and transaction processing. Traditional algorithms like **Bubble Sort** and **Insertion Sort** operate in  $O(N^2)$  time, making them infeasible for large datasets.

**Merge Sort** is a **divide-and-conquer algorithm** that:

1. Divides the array into two halves.
2. Recursively sorts both halves.
3. Merges the two sorted halves into a single sorted array.

### Key properties of Merge Sort:

- Stable sorting algorithm (preserves order of equal timestamps).
- Guaranteed time complexity of  $O(N \log N)$ .
- Suitable for large datasets, though it requires  $O(N)$  extra space.
- Performs well on **linked lists** and **external sorting** (data too large for RAM).

By comparing Merge Sort with **Quick Sort**, **Bubble Sort**, and **Insertion Sort**, we can observe its advantages in scalability and predictability.

## Algorithm:

### Merge Sort Algorithm

#### Steps:

1. If the list has one element or is empty  $\rightarrow$  return (base case).
2. Divide the list into two halves (left and right).
3. Recursively apply Merge Sort on each half.
4. Merge the two sorted halves into one sorted array.

#### Pseudocode:

```
void merge(int arr[],int n,int start,int mid,int end)
```

```
{
    int i=start;
    int j=mid+1;
    int k=0;
    int temp[n];
    while(i<=mid && j<=end)
    {
        if(arr[i]<arr[j])
        {
            temp[k]=arr[i];
            i++;
        }
        else
        {
            temp[k]=arr[j];
            j++;
        }
        k++;
    }
}
```

```

    if(i>mid)
    {
        while(j<=end)
        {
            temp[k]=arr[j];
            j++;
            k++;
        }
    }
    else
    {
        while(i<=mid)
        {
            temp[k]=arr[i];
            i++;
            k++;
        }
    }
    for(int i=0;i<k;i++)
    {
        arr[start+i]=temp[i];
    }
}

void mergeSort(int arr[],int n,int start,int end)
{
    if(start<end)
    {
        int mid=(start+end)/2;
        mergeSort(arr,n,start,mid);
        mergeSort(arr,n,mid+1,end);
        merge(arr,n,start,mid,end);
    }
}

```

**Time Complexity:**

- **Merge Sort:**

- Best Case:  $O(N \log N)$
- Average Case:  $O(N \log N)$
- Worst Case:  $O(N \log N)$
- Space Complexity:  $O(N)$  (auxiliary arrays).

### ● Comparison with Traditional Sorting Algorithms:

- **Bubble Sort:**  $O(N^2)$  — inefficient for large datasets.
- **Insertion Sort:**  $O(N^2)$  average,  $O(N)$  best (nearly sorted data).
- **Selection Sort:**  $O(N^2)$ .
- **Quick Sort:**  $O(N \log N)$  average,  $O(N^2)$  worst case (unbalanced partitions).
- **Merge Sort:**  $O(N \log N)$  guaranteed, stable.

### Why Merge Sort is better for large datasets:

- Predictable  $O(N \log N)$  runtime regardless of input order.
- Well-suited for linked lists and external sorting (huge datasets that don't fit in memory).
- Handles up to **1 million orders efficiently** ( $\approx 20$  million comparisons at most).

**Conclusion:** The Merge Sort algorithm efficiently sorts customer orders based on timestamps, making it highly suitable for large-scale applications such as e-commerce order management. Unlike quadratic algorithms (Bubble, Insertion, Selection), Merge Sort scales well with millions of records. Its guaranteed  $O(N \log N)$  performance and stability make it superior for high-volume transaction systems. While Quick Sort may outperform in practice on average, Merge Sort avoids worst-case pitfalls and provides consistent performance.

### Questions:

1. Why is Merge Sort considered a stable sorting algorithm?

ANS : **Merge Sort is stable** because when it merges two parts of the array, if two numbers are the same, it keeps them in the same order they appeared in the original list.

### Example:

Original list:

[ (5, A), (3, B), (5, C), (2, D) ]

After Merge Sort:

[ (2, D), (3, B), (5, A), (5, C) ]

- Notice **A comes before C**, just like in the original list.

2. How does Merge Sort compare with traditional sorting methods like Bubble Sort and Insertion Sort?

ANS :

- **Merge Sort** is fast for large data ( $O(n \log n)$ ) but uses extra space.
- **Bubble Sort** is simple but slow ( $O(n^2)$ ).

- **Insertion Sort** is simple and good for small or nearly sorted data.
- For big datasets → Use **Merge Sort**
- For small or almost sorted data → Use **Insertion Sort**
- For learning purposes → **Bubble Sort** is easy to understand but not efficient.