Pimpri Chinchwad Education Trust's
Pimpri Chinchwad College of Engineering (PCCoE)
(An Autonomous Institute)
Affiliated to Savitribai Phule Pune University(SPPU)
ISO 21001:2018 Certified by TUV

Assignment -1

· **Name: Khushalsing Pawar**
· **Branch: Information Technology Division: B PRN No.: 124B2F011**
· **Course Name : Design and Analysis of Algorithms**

1. Problem Statement:

Design and implement a sorting algorithm using Merge Sort to efficiently arrange customer orders based on their timestamps. The solution should handle a large dataset (up to 1 million orders) with minimal computational overhead. Additionally, analyze the time complexity and compare it with traditional sorting techniques.

2. Course Objective:

2.1. To know the basics of computational complexity of various algorithms.

2.2. To select appropriate algorithm design strategies to solve real-world problems.

3. Course Outcome:

3.1. Analyze the asymptotic performance of algorithms

3.2. Solve computational problems by applying suitable paradigms of Divide and Conquer or Greedy methodologies

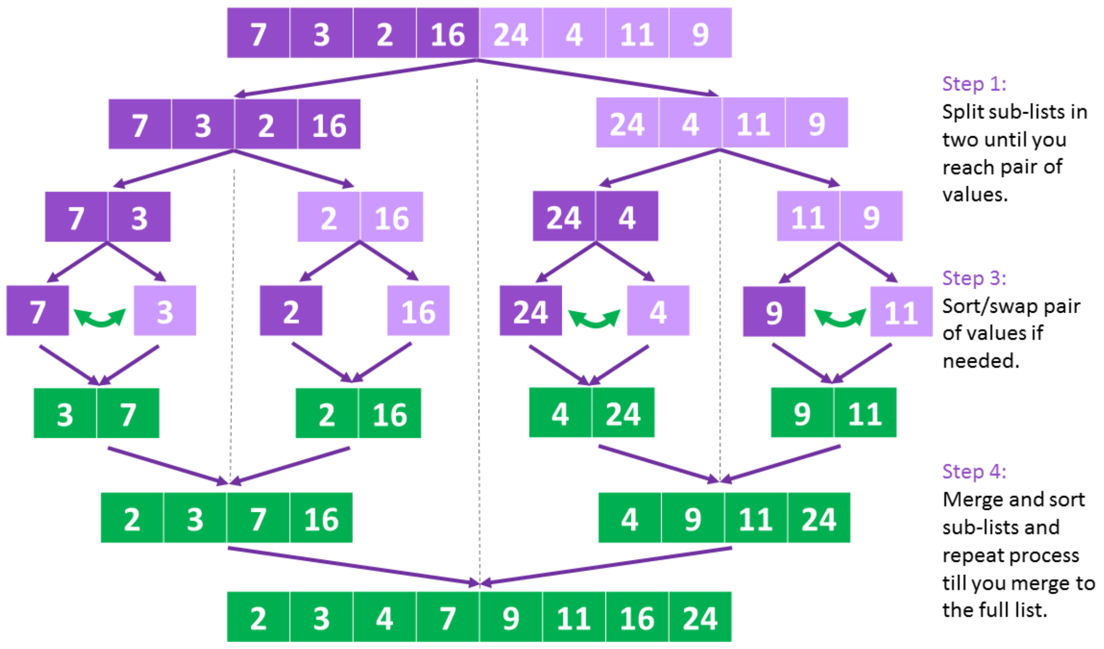
4. Theory:

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide and conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.

How does Merge Sort work?

1. **Divide:** Keep dividing the list or array into two equal halves again and again, until each part has only one element left.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.

3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.


7 3 2 16 24 4 11 9
7 3 2 16
24 4 11 9
7 3
2 16
24 4
11 9
7
3
2
16
24
4
9
11
3 7
2 16
4 24
9 11
2 3 7 16
4 9 11 24
2 3 4 7 9 11 16 24
Step 1:
Split sub-lists in two until you reach pair of values.
Step 3:
Sort/swap pair of values if needed.
Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Complexity Analysis of Merge Sort

How Merge Sort Works

- **Divide step**: The array is recursively divided into two halves until each subarray has one element.
    - Number of times we can divide an array of size n = $\log_2(n)$ (height of recursion tree).
- **Merge step**: At each level of recursion, all n elements are merged.
- **Total work** = number of levels × work per level = $\log_2(n) \times n = O(n \log n)$.

Analysis of Merge Sort Time Complexity

Best Case Time Complexity of Merge Sort

- Even if the array is already sorted, Merge Sort still **divides** and **merges** every element.
- Merging sorted subarrays still requires scanning through all elements.
  Best Case = O(n log n)

Average Case Time Complexity

- For randomly arranged data, the algorithm still divides into halves and merges them.
- On average, the number of comparisons during merging is proportional to n.
  Average Case = O(n log n)

Worst Case Time Complexity

- Even if the array is sorted in **descending order** (worst arrangement for comparisons), Merge Sort still performs the same divide-and-merge procedure.
- Comparisons may be maximum, but still bounded by O(n log n).
  Worst Case = O(n log n)

Analysis of Merge Sort Space Complexity

In merge sort, all elements are copied into an **auxiliary array** of size N, where N is the **number of elements present in the unsorted array**. Hence, the space complexity for **Merge Sort** is O(N).

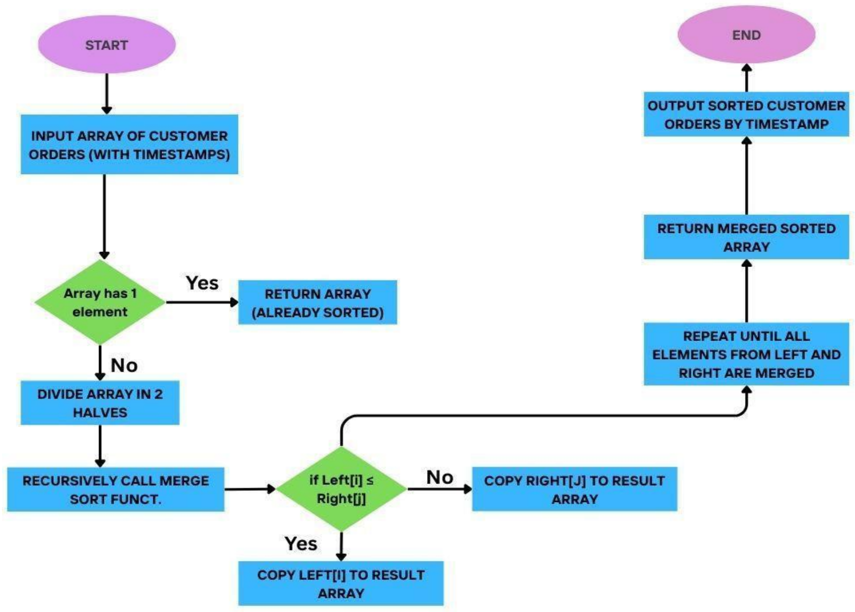
Merge Sort Algorithm

Step 1: If it is only one element in the list, consider it already sorted, so return.
Step 2: Divide the list recursively into two halves until it can't no more be divided.
Step 3: Merge the smaller lists into new list in sorted order.

Pseudocode

```
procedure mergesort( var a as array )
  if ( n == 1 ) return a
    var l1 as array = a[0] ... a[n/2]
    var l2 as array = a[n/2+1] ... a[n]
    l1 = mergesort( l1 )
    l2 = mergesort( l2 )
    return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while
  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while
  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while
  return c
end procedure
```

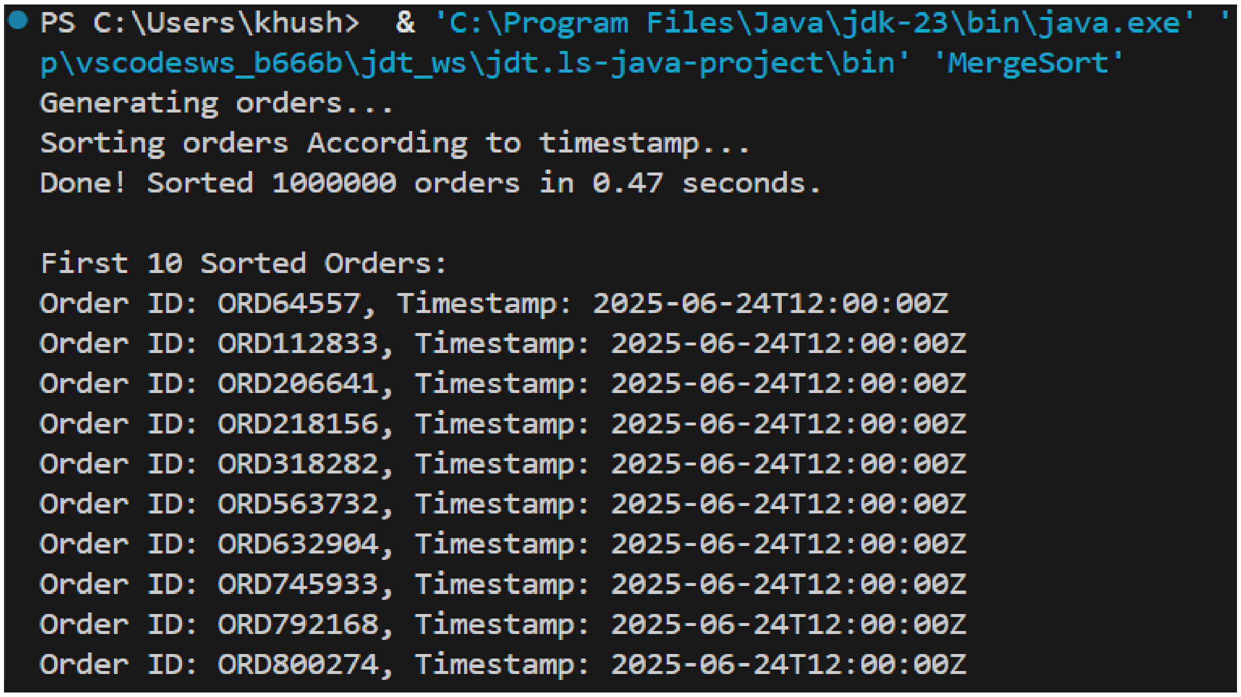
5. Implementation:


START
INPUT ARRAY OF CUSTOMER ORDERS (WITH TIMESTAMPS)
Array has 1 element
Yes
RETURN ARRAY (ALREADY SORTED)
No
DIVIDE ARRAY IN 2 HALVES
RECURSIVELY CALL MERGE SORT FUNCT.
if Left[i] ≤ Right[j]
No
COPY RIGHT[J] TO RESULT ARRAY
Yes
COPY LEFT[I] TO RESULT ARRAY
REPEAT UNTIL ALL ELEMENTS FROM LEFT AND RIGHT ARE MERGED
RETURN MERGED SORTED ARRAY
OUTPUT SORTED CUSTOMER ORDERS BY TIMESTAMP
END

6. Code:

```
import java.util.*;
import java.text.SimpleDateFormat;

class Order implements Comparable<Order> {
    String orderId;
    long timestamp;

    public Order(String orderId, long timestamp) {
        this.orderId = orderId;
        this.timestamp = timestamp;
    }
    @Override
    public int compareTo(Order other) {
        return Long.compare(this.timestamp, other.timestamp);
    }
}

public class MergeSort {
    private static final int NUM_ORDERS = 1000000;

    public static List<Order> generateSampleOrders(int n) {
        List<Order> orders = new ArrayList<>(n);

        Calendar calendar = Calendar.getInstance(TimeZone.getTimeZone("UTC"));
        calendar.set(2025, Calendar.JUNE, 24, 12, 0, 0);
        calendar.set(Calendar.MILLISECOND, 0);
        long baseTime = calendar.getTimeInMillis() / 1000;

        Random random = new Random();

        for (int i = 0; i < n; i++) {
            int randomMinutes = random.nextInt(100000);
            long timestamp = baseTime + (randomMinutes * 60);
            orders.add(new Order("ORD" + (i + 1), timestamp));
        }

        return orders;
    }

    public static void mergeSort(List<Order> orders) {
        if (orders.size() <= 1) return;

        int mid = orders.size() / 2;
        List<Order> left = new ArrayList<>(orders.subList(0, mid));
        List<Order> right = new ArrayList<>(orders.subList(mid, orders.size()));

        mergeSort(left);
        mergeSort(right);

        merge(orders, left, right);
    }

    private static void merge(List<Order> result, List<Order> left, List<Order> right) {
        int i = 0, j = 0, k = 0;

        while (i < left.size() && j < right.size()) {
            if (left.get(i).timestamp <= right.get(j).timestamp) {
                result.set(k++, left.get(i++));
            } else {
                result.set(k++, right.get(j++));
            }
        }



        while (i < left.size()) {
            result.set(k++, left.get(i++));
        }

        while (j < right.size()) {
            result.set(k++, right.get(j++));
        }
    }

    public static void printFirstNOrders(List<Order> orders, int n) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'Z'");
        sdf.setTimeZone(TimeZone.getTimeZone("UTC"));

        int limit = Math.min(n, orders.size());
        for (int i = 0; i < limit; i++) {
            Order order = orders.get(i);
            Date date = new Date(order.timestamp * 1000L);
            System.out.println("Order ID: " + order.orderId +
                    ", Timestamp: " + sdf.format(date));
        }
    }

    public static void main(String[] args) {
        System.out.println("Generating orders...");
        List<Order> orders = generateSampleOrders(NUM_ORDERS);

        System.out.println("Sorting orders According to timestamp...");
        long startTime = System.nanoTime();
        mergeSort(orders);
        long endTime = System.nanoTime();

        double timeTaken = (endTime - startTime) / 1_000_000_000.0;
        System.out.printf("Done! Sorted %d orders in %.2f seconds.%n",
                NUM_ORDERS, timeTaken);

        System.out.println("\nFirst 10 Sorted Orders:");
        printFirstNOrders(orders, 10);
    }
}
```

7. Output:



```
PS C:\Users\khush>  & 'C:\Program Files\Java\jdk-23\bin\java.exe' '
p\vscodesws_b666b\jdt_ws\jdt.ls-java-project\bin' 'MergeSort'
Generating orders...
Sorting orders According to timestamp...
Done! Sorted 1000000 orders in 0.47 seconds.

First 10 Sorted Orders:
Order ID: ORD64557, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD112833, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD206641, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD218156, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD318282, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD563732, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD632904, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD745933, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD792168, Timestamp: 2025-06-24T12:00:00Z
Order ID: ORD800274, Timestamp: 2025-06-24T12:00:00Z
```

8. Conclusion:

Merge Sort is a highly efficient and stable sorting algorithm that organizes customer orders based on their timestamps with **O(n log n)** complexity. Compared to traditional algorithms like Bubble Sort or Insertion Sort, which are too slow for large inputs, Merge Sort can easily handle up to **1 million records** without performance issues. Its ability to maintain the correct order and scalability makes it an excellent choice for real-world applications such as **e-commerce, order tracking, and scheduling systems**.

9. Questions: