



**Pimpri Chinchwad Education Trust's
Pimpri Chinchwad college of Engineering**

Assignment No: 1

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.

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 \times 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.

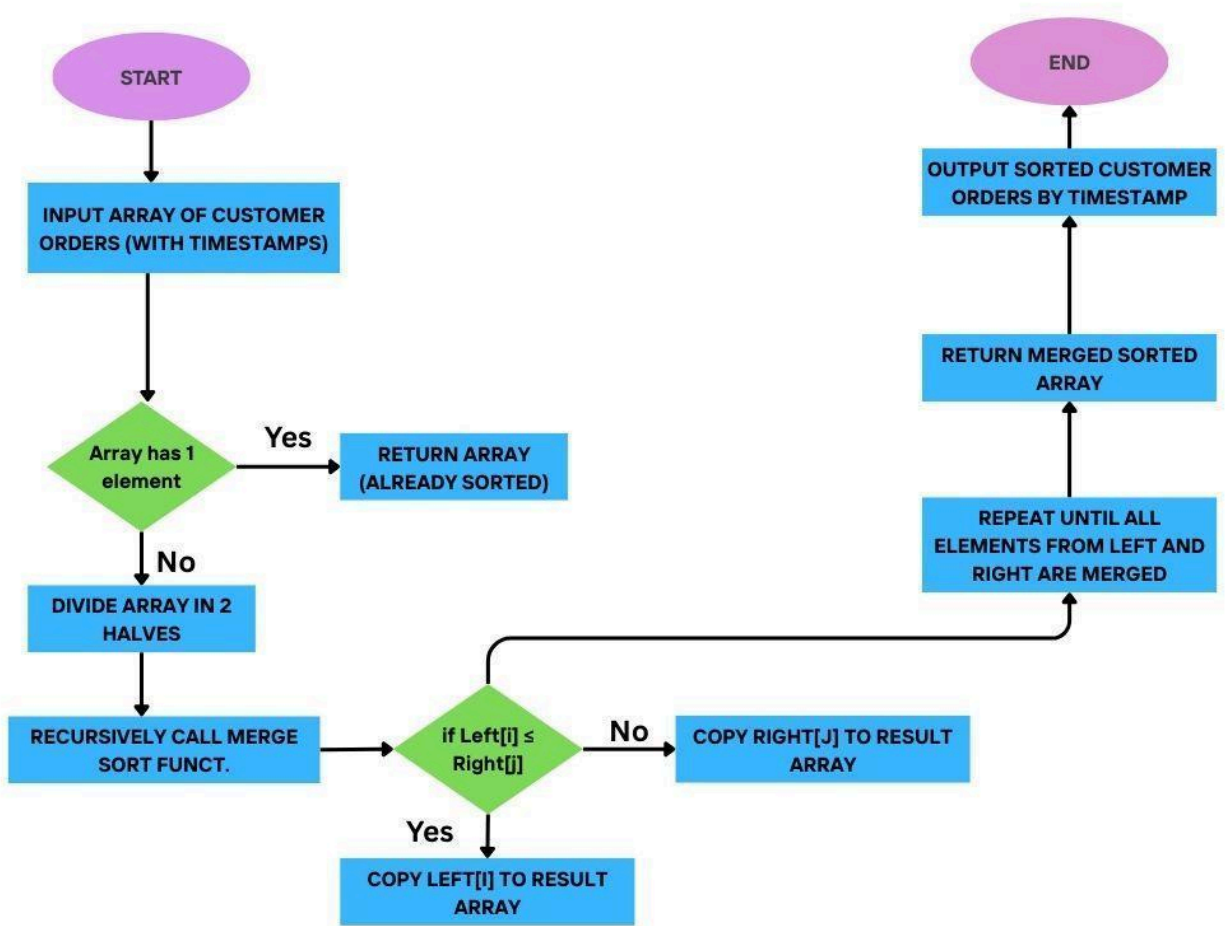
Step 3: Merge the smaller lists into new list in sorted order.

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:



6. Output:

Customer order timestamps:
[12:30, 09:15, 14:45, 10:00]

Step 1: Divide

- Split into halves → [12:30, 09:15] and [14:45, 10:00]
- Further split → [12:30], [09:15], [14:45], [10:00]

Step 2: Merge & Sort

- Merge [12:30] and [09:15] → [09:15, 12:30]
- Merge [14:45] and [10:00] → [10:00, 14:45]

- Merge final two halves → [09:15, 10:00, 12:30, 14:45]

Final Output:

Orders sorted by timestamp → [09:15, 10:00, 12:30, 14:45]

Code:

```
#include <iostream>
```

```
#include <vector>
```

```
#include <ctime>
```

```
#include <cstdlib>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
size_t max_auxiliary_space = 0;
```

```
struct Order {  
    int order_id;  
    time_t timestamp;  
};
```

```
void merge(vector<Order>& orders, int left, int mid, int right) {  
    int n1 = mid - left + 1;  
    int n2 = right - mid;  
  
    vector<Order> L(orders.begin() + left, orders.begin() + mid + 1);  
    vector<Order> R(orders.begin() + mid + 1, orders.begin() + right + 1);
```

```
size_t current_aux_space = (L.size() + R.size()) * sizeof(Order);
```

```
if (current_aux_space > max_auxiliary_space) {  
    max_auxiliary_space = current_aux_space;  
}
```

```
int i = 0, j = 0, k = left;
```

```
while (i < n1 && j < n2) {  
    if (L[i].timestamp <= R[j].timestamp) {  
        orders[k++] = L[i++];  
    } else {  
        orders[k++] = R[j++];  
    }  
}
```

```
while (i < n1) {  
    orders[k++] = L[i++];  
}
```

```
while (j < n2) {  
    orders[k++] = R[j++];  
}
```

```
}
```

```
void merge_sort(vector<Order>& orders, int left, int right) {  
    if (left < right) {
```

```

    int mid = left + (right - left) / 2;

    merge_sort(orders, left, mid);

    merge_sort(orders, mid + 1, right);

    merge(orders, left, mid, right);

}

}

```

```

void generate_sample_orders(vector<Order>& orders, int n) {

    struct tm base_time_tm = {};

    base_time_tm.tm_year = 2025 - 1900;

    base_time_tm.tm_mon = 5;

    base_time_tm.tm_mday = 24;

    base_time_tm.tm_hour = 12;

    time_t base_time = mktime(&base_time_tm);

    orders.resize(n);

    for (int i = 0; i < n; i++) {

        int random_minutes = rand() % 100000;

        orders[i].timestamp = base_time + (random_minutes * 60);

        orders[i].order_id = i + 1;

    }

}

```

```

void print_first_n_orders(const vector<Order>& orders, int n) {

    char time_str[30];

    cout << "\n--- First 5 Sorted Orders ---\n";

```

```

int limit = min(n, static_cast<int>(orders.size()));

for (int i = 0; i < limit; i++) {

    struct tm time_str;
    strftime(time_str, sizeof(time_str), "%Y-%m-%dT%H:%M:%SZ",
gmtime(&orders[i].timestamp));

    cout << "Order ID: " << left << setw(10) << orders[i].order_id
        << "| Timestamp: " << time_str << endl;

}
}

```

```

int main() {

    srand(time(nullptr));

    int num_orders;

    cout << "Enter the number of orders to generate and sort: ";

    cin >> num_orders;

    if (cin.fail() || num_orders <= 0) {

        cout << "Error: Invalid input. Please enter a positive number." << endl;

        return 1;

    }
}

```

```

vector<Order> orders;

```

```

cout << "\nGenerating " << num_orders << " orders...\n";

generate_sample_orders(orders, num_orders);

```

```

cout << "Sorting orders using Merge Sort...\n";

clock_t start = clock();

if (!orders.empty()) {
    merge_sort(orders, 0, orders.size() - 1);
}

clock_t end = clock();

cout << "Done!\n";

double time_taken = static_cast<double>(end - start) / CLOCKS_PER_SEC;

cout << "\nTime taken to sort " << num_orders << " orders: "
    << fixed << setprecision(2) << time_taken << " seconds.\n";

print_first_n_orders(orders, 5);

cout << "\n--- Space Complexity Report ---\n";

cout << "Size of one Order object: " << sizeof(Order) << " bytes\n";

cout << "Total input size: " << orders.size() * sizeof(Order) << " bytes\n";

cout << "Peak auxiliary space used: " << max_auxiliary_space << " bytes\n";

return 0;
}

```

Output:

Enter the number of orders to generate and sort: 1000000

Generating 1000000 orders...

Sorting orders using Merge Sort...

Done!

Time taken to sort 1000000 orders: 0.67 seconds.

--- First 5 Sorted Orders ---

Order ID: 274005 | Timestamp: 2025-06-24T12:00:00Z

Order ID: 331391 | Timestamp: 2025-06-24T12:00:00Z

Order ID: 345713 | Timestamp: 2025-06-24T12:00:00Z

Order ID: 390635 | Timestamp: 2025-06-24T12:00:00Z

Order ID: 392329 | Timestamp: 2025-06-24T12:00:00Z

--- Space Complexity Report ---

Size of one Order object: 16 bytes

Total input size: 16000000 bytes

Peak auxiliary space used: 16000000 bytes

7. 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**.

