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

Department: Information Technology **Academic Year**: 2025-26 **Semester:** V

DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

Name : Avishkar Jadhav **PRN** : 123B1F032

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 → 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);
    }
}
```

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define NUM_ORDERS 1000000  // Use 10000 for testing

typedef struct {
    char order_id[20];
    time_t timestamp;
} Order;

// Generate random orders
void generate_sample_orders(Order *orders, int n) {
    struct tm base_time = {0};
    base_time.tm_year = 2025 - 1900;
    base_time.tm_mon = 5;     // June (0-based)
    base_time.tm_mday = 24;
    base_time.tm_hour = 12;

    time_t base = mktime(&base_time);

    for (int i = 0; i < n; i++) {
        int random_minutes = rand() % 100000;  // up to ~70 days
        orders[i].timestamp = base + (random_minutes * 60);
        sprintf(orders[i].order_id, "ORD%d", i + 1);
    }
}

// Merge Sort functions
void merge(Order *orders, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    Order *L = malloc(n1 * sizeof(Order));
    Order *R = malloc(n2 * sizeof(Order));

    for (int i = 0; i < n1; i++) L[i] = orders[left + i];
    for (int j = 0; j < n2; j++) R[j] = orders[mid + 1 + j];

    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++];

    free(L);
    free(R);
}

void merge_sort(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);
    }
}

// Print first 5 orders
void print_first_n_orders(Order *orders, int n) {
    char time_str[30];
    for (int i = 0; i < n; i++) {
        struct tm *tm_info = gmtime(&orders[i].timestamp);
        strftime(time_str, sizeof(time_str), "%Y-%m-%dT%H:%M:%SZ", tm_info);
        printf("Order ID: %s, Timestamp: %s\n", orders[i].order_id, time_str);
    }
}

int main() {
    srand(time(NULL));

    Order *orders = malloc(NUM_ORDERS * sizeof(Order));

    printf("Generating orders...\n");
    generate_sample_orders(orders, NUM_ORDERS);

    printf("Sorting orders by timestamp...\n");
    clock_t start = clock();
    merge_sort(orders, 0, NUM_ORDERS - 1);
    clock_t end = clock();

    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Done! Sorted %d orders in %.2f seconds.\n", NUM_ORDERS, time_taken);
    printf("\n📄 First 5 Sorted Orders:\n");
    print_first_n_orders(orders, 5);


    free(orders);
    return 0;
}
```

OUTPUT

```
Generating orders...
Sorting orders by timestamp...
Done! Sorted 1000000 orders in 2.84 seconds.
 First 5 Sorted Orders:
Order ID: ORD50213, Timestamp: 2025-06-24T12:03:00Z
Order ID: ORD731002, Timestamp: 2025-06-24T12:04:00Z
Order ID: ORD18922, Timestamp: 2025-06-24T12:05:00Z
Order ID: ORD458721, Timestamp: 2025-06-24T12:05:00Z
Order ID: ORD820134, Timestamp: 2025-06-24T12:06:00Z
```

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** (≈ 20 million comparisons at most).

Questions:

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

A sorting algorithm is **stable** when it preserves the relative order of equal elements in the list. That means, if two elements have the same value and one comes before the other in the original list, it should remain before the other after sorting.

Merge Sort is stable because:

1. During the merge step, when it compares elements from the left and right subarrays:
    - If the elements are equal, it **chooses the element from the left subarray first**.
    - This ensures that elements with equal value retain their original order.

Example:

Original array:
[(5, 'a'), (3, 'b'), (5, 'c'), (2, 'd')]
Here, (5, 'a') and (5, 'c') have the same value but different tags.

After merge sort:
[(2, 'd'), (3, 'b'), (5, 'a'), (5, 'c')]

You can see that (5, 'a') still comes before (5, 'c'), which confirms that the algorithm is stable.

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

Feature	**Merge Sort**	**Bubble Sort**	**Insertion Sort**
Time Complexity	Merge Sort takes O(n log n) time in all cases.	Bubble Sort takes O(n^2) time in most cases and O(n) if already sorted.	Insertion Sort takes O(n^2) time in most cases and O(n) if already sorted.
Stability	Merge Sort is stable, so equal elements keep their order.	Bubble Sort is stable, so equal elements stay in the same order.	Insertion Sort is stable, so equal elements stay in the same order.

Working Method	Merge Sort divides the array and merges sorted parts.	Bubble Sort swaps adjacent elements if they are in the wrong order.	Insertion Sort inserts each element at its correct place.
Space Usage	Merge Sort needs extra space for temporary arrays.	Bubble Sort works in-place, using no extra space.	Insertion Sort works in-place, using no extra space.
Best Use	Merge Sort is best for large datasets.	Bubble Sort is good for small datasets or learning.	Insertion Sort is good for small or nearly sorted datasets.

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.