|  |  |  |
| --- | --- | --- |
|  | 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** | | |

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);

}

}

# 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(N²) — inefficient for large datasets.
  + **Insertion Sort:** O(N²) average, O(N) best (nearly sorted data).
  + **Selection Sort:** O(N²).
  + **Quick Sort:** O(N log N) average, O(N²) 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?

# Merge Sort is considered stable because it preserves the original relative order of elements with equal values. This property is guaranteed during the merge step. When merging two sorted subarrays (left and right), if an element in the left subarray has the same value as an element in the right subarray, the algorithm is designed to take the element from the left subarray first. This ensures that elements that appeared earlier in the original input array maintain their order relative to other elements of the same value.

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

# Merge Sort is significantly more efficient for large datasets compared to Bubble Sort and Insertion Sort. Here's a direct comparison:

# Time Complexity: Merge Sort has a consistent time complexity of O(NlogN) for all cases (best, average, and worst). In contrast, Bubble Sort and Insertion Sort have an average and worst-case time complexity of O(N2). This quadratic growth makes them extremely slow for large inputs.

# Scalability: Due to its superior time complexity, Merge Sort scales effectively for large datasets, like sorting one million customer orders. Bubble Sort and Insertion Sort are considered infeasible for such tasks as the computation time would be excessively long.

# Performance: While Insertion Sort can be faster for very small or nearly sorted arrays (with a best-case complexity of O(N)), Merge Sort's performance is predictable and reliable across all types of input data, making it a better general-purpose choice for large-scale sorting.

**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.

**Code:**

#include <iostream>

#include <vector>

#include <string>

#include <ctime>

#include <cstdlib>

#include <iomanip>

#include <sstream>

#include <chrono>

using namespace std;

#define NUM\_ORDERS 1000000

struct Order {

string order\_id;

time\_t timestamp;

};

// Generate random orders

void generate\_sample\_orders(vector<Order>& orders, int n) {

tm base\_time = {};

base\_time.tm\_year = 2025 - 1900;

base\_time.tm\_mon = 5; // June

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

Order order;

order.timestamp = base + (random\_minutes \* 60);

order.order\_id = "ORD" + to\_string(i + 1);

orders.push\_back(order);

}

}

// Merge function

void merge(vector<Order>& orders, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

vector<Order> L(n1), R(n2);

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++];

}

// Merge Sort

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);

}

}

// Print orders

void print\_first\_n\_orders(const vector<Order>& orders, int n) {

char buffer[30];

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

tm\* tm\_info = gmtime(&orders[i].timestamp);

strftime(buffer, sizeof(buffer), "%Y-%m-%dT%H:%M:%SZ", tm\_info);

cout << "Order ID: " << orders[i].order\_id

<< ", Timestamp: " << buffer << "\n";

}

}

int main() {

srand(time(NULL));

vector<Order> orders;

orders.reserve(NUM\_ORDERS);

cout << "Generating orders...\n";

generate\_sample\_orders(orders, NUM\_ORDERS);

cout << "Sorting orders by timestamp...\n";

auto start = chrono::high\_resolution\_clock::now();

merge\_sort(orders, 0, orders.size() - 1);

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> time\_taken = end - start;

cout << "Done! Sorted " << NUM\_ORDERS

<< " orders in " << time\_taken.count() << " seconds.\n";

cout << "\n First " << NUM\_ORDERS << " Sorted Orders:\n";

print\_first\_n\_orders(orders, NUM\_ORDERS);

return 0;

}

**Output:**

Generating orders...

Sorting orders by timestamp...

Done! Sorted 20 orders in 0.00 seconds.

First 20 Sorted Orders:

Order ID: ORD18, Timestamp: 2025-06-25T02:24:00Z

Order ID: ORD7, Timestamp: 2025-06-25T18:51:00Z

Order ID: ORD13, Timestamp: 2025-06-27T04:27:00Z

Order ID: ORD3, Timestamp: 2025-06-28T21:18:00Z

Order ID: ORD1, Timestamp: 2025-06-30T15:06:00Z

Order ID: ORD9, Timestamp: 2025-07-01T07:32:00Z

Order ID: ORD11, Timestamp: 2025-07-03T12:14:00Z

Order ID: ORD14, Timestamp: 2025-07-04T09:59:00Z

Order ID: ORD20, Timestamp: 2025-07-05T20:40:00Z

Order ID: ORD2, Timestamp: 2025-07-07T01:47:00Z

Order ID: ORD5, Timestamp: 2025-07-09T11:23:00Z

Order ID: ORD12, Timestamp: 2025-07-10T03:56:00Z

Order ID: ORD8, Timestamp: 2025-07-11T19:30:00Z

Order ID: ORD4, Timestamp: 2025-07-13T06:42:00Z

Order ID: ORD15, Timestamp: 2025-07-14T18:15:00Z

Order ID: ORD6, Timestamp: 2025-07-16T04:22:00Z

Order ID: ORD19, Timestamp: 2025-07-17T08:51:00Z

Order ID: ORD10, Timestamp: 2025-07-18T15:03:00Z

Order ID: ORD16, Timestamp: 2025-07-19T11:27:00Z

Order ID: ORD17, Timestamp: 2025-07-20T23:08:00Z0:41:00Z