Koneru Lakshmaiah Education Foundation
(Deemed to be University estd. u/s. 3 of the UGC Act, 1956)
Off-Campus: Bachupally-Gandimaisamma Road, Bowrampet, Hyderabad, Telangana - 500 043.
Phone No: 7815926816, www.klh.edu.in

Department of Computer Science and Engineering

2025-2026
Odd Semester

DESIGN AND ANALYSIS ALGORITHMS (24CS2203)

ALM – PROJECT BASED LEARNING

MERGE SORT IN AIRLINE TICKET MANAGEMENT

T Sai Shreeja	**2420030631**
G. Divya Sree	**2420030287**

COURSE INSTRUCTOR

Dr. J Sirisha Devi
Professor

Department of Computer Science and Engineering

MERGE SORT IN AIRLINE TICKET MANAGEMENT

Brief Description

In the modern travel industry, efficiency in data processing plays a crucial role in ensuring a smooth customer experience. Airline ticket management systems are at the core of this operation — they handle vast volumes of flight data every second, including flight schedules, ticket prices, seat availability, and passenger details. As travelers search for flights, the system must retrieve and display relevant results in a matter of milliseconds.

One of the most essential operations behind this efficiency is sorting. Sorting flight schedules enables faster search results, easy comparisons, and efficient memory handling. Among various sorting algorithms, Merge Sort stands out for its stability, predictable performance, and ability to handle large datasets, even those stored across multiple servers.

Case Example:

Imagine an airline booking system like *IndiGo's* or *Air India's* central reservation database. Every day, the system manages data for thousands of flights worldwide. Each flight record contains:

- Flight number (e.g., AI203)
- Departure time
- Arrival time
- Ticket price
- Duration
- Seat availability

When a customer searches for flights from *Hyderabad to Delhi*, the system fetches all matching flights. However, this data is often unsorted — flights may appear in random order, pulled from different regional servers.

To enhance the user experience, the system must display flights sorted by departure time or ticket price. Here's where Merge Sort plays its role.

The merge sort algorithm:

1. Divides the dataset of flights into smaller subsets.
2. Sorts each subset individually.
3. Merges the sorted subsets into a single sorted list.

For Example:

Unsorted List (by Departure Time)

IndiGo 18:30

Air India 06:00

Vistara 13:45

SpiceJet 09:20

After Merge Sort:

Air India 06:00

SpiceJet 09:20

Vistara 13:45

IndiGo 18:30

This sorting ensures that when a passenger searches for flights, the display order feels intuitive and quick. Merge Sort's efficiency (O(n log n)) ensures that even large datasets — such as 50,000 flight entries — can be handled smoothly.

Merge Sort Fits Airline Systems Because:

1. Stable Sorting: Keeps records consistent when two flights have the same departure time.
2. Predictable Performance: Maintains efficiency in both best and worst cases.
3. External Sorting Capability: Works well with data stored across multiple memory locations or servers — ideal for airline data distributed in databases.
4. Parallelism: Merge sort's divide-and-conquer approach allows parallel execution, improving response time during peak hours.

Thus, merge sort helps airline ticket management systems deliver fast, reliable, and user-friendly booking experiences — a cornerstone of modern air travel operations.

ALGORITHM

Below is the algorithm of the Merge Sort

```
Algorithm FlightMergeSort(Flights)
   Input: A list of flight records (FlightNumber, DepartureTime, ArrivalTime, Price)
   Output: Sorted list of flights by DepartureTime

   1. If the number of flights ≤ 1, return Flights.
   2. Find the middle index:
        mid ← length(Flights) / 2
   3. Split the list into two halves:
        Left ← Flights[0 ... mid-1]
        Right ← Flights[mid ... end]
   4. Recursively call FlightMergeSort on both halves:
        Left ← FlightMergeSort(Left)
        Right ← FlightMergeSort(Right)
   5. Merge the two sorted halves:
        return MergeFlights(Left, Right)
end Algorithm


Algorithm MergeFlights(Left, Right)
   1. Initialize an empty list SortedFlights
   2. Set i ← 0, j ← 0
   3. While i < length(Left) and j < length(Right):
        a. If Left[i].DepartureTime ≤ Right[j].DepartureTime:
             Add Left[i] to SortedFlights
             i ← i + 1
           Else:
             Add Right[j] to SortedFlights
             j ← j + 1
   4. Add any remaining flights from Left (if any) to SortedFlights
   5. Add any remaining flights from Right (if any) to SortedFlights
   6. Return SortedFlights
end Algorithm
```

Step-by-Step Explanation

Step 1: Input and Objective

The algorithm takes a list of flight records as input.
Each record typically includes:

- Flight Number
- Departure Time
- Arrival Time
- Ticket Price

The goal is to arrange the flights in ascending order of departure time, so that users can view schedules in a logical, time-based sequence.

Step 2: Base Case (Lines 1–2)

If there's only one flight or no flights in the list, it's already sorted — so the algorithm returns it immediately.
This base condition ensures that the recursion will eventually stop.

Step 3: Divide Phase (Lines 2–3)

The list is divided into two halves:

- Left half: contains the first half of the flights
- Right half: contains the remaining flights

Example:

Flights = [AI203, SG501, VI112, 6E401]

Left = [AI203, SG501]

Right = [VI112, 6E401]

This division continues recursively until each list contains just one flight.

Step 4: Recursive Sorting (Line 4)

Each smaller list (Left and Right) is then passed again into the same FlightMergeSort algorithm.
This means the algorithm keeps dividing the list until every subset has just one element.
At that point, those single-flight lists are considered sorted.

Step 5: Merge Phase (Line 5)

Once the lists are reduced to single elements, the algorithm begins to merge them back together — but in sorted order.
The merging is handled by the MergeFlights subroutine.

Step 6: Comparison and Merging (MergeFlights)

- Two pointers (i for Left, j for Right) are used to track the current position in both lists.
- The flights at each pointer are compared by DepartureTime.
- Whichever flight has the earlier departure is added to the new list SortedFlights.
- The pointer for that list (Left or Right) is then advanced.

This process repeats until all elements from both lists are merged into SortedFlights.

Step 7: Handling Remaining Flights

Once one list (Left or Right) is exhausted, any remaining flights from the other list are directly added to the result — they're already sorted.

Step 8: Final Output

The merged and fully sorted list is returned as the final output.
For example:

Input (Unsorted):

[IndiGo 18:30, Air India 06:00, Vistara 13:45, SpiceJet 09:20]

Output (Sorted by Departure Time):

[Air India 06:00, SpiceJet 09:20, Vistara 13:45, IndiGo 18:30]

TIME COMPLEXITY

Merge sort's time complexity remains consistent across all cases (best, average, and worst) because it always divides the dataset in half and merges them in linear time.

Let *n* be the number of flights in the list.

Step 1: Divide Phase

Each recursive call splits the list into two halves until each sublist has one element. Number of levels in the recursion tree = $\mathbf{\log_2(n)}$.

Step 2: Merge Phase

At each level, the merging process takes **O(n)** time because every element must be compared and merged once.

Total Time Complexity

$$T(n) = 2T(n/2) + O(n)$$

Using the **Master Theorem**,

$$T(n) = O(nlogn)$$

Case Wise Analysis:

Case	Condition	Time Complexity
Best Case	Already sorted flight list	O(n log n)
Average Case	Random flight order	O(n log n)
Worst Case	Reverse sorted list	O(n log n)

Even in the worst scenario, the complexity doesn't degrade to $O(n^2)$ like bubble or insertion sort — making merge sort ideal for large airline databases.

SPACE COMPLEXITY

Merge Sort requires additional memory for temporary arrays during the merging process.

Let *n* be the total number of flight records.

Detailed Calculation:

1. During the merge operation, two temporary arrays (L and R) are created to hold divided data.
2. Each recursive call creates new arrays until the entire list is broken down.
3. However, the total extra space used at any point is proportional to n.

$$Space\ Complexity = \boldsymbol{O}(\boldsymbol{n})$$

Breakdown:

- Input Array: O(n)
- Temporary Arrays (L and R): O(n)
- Recursive Stack Space: O(log n)

Therefore, total space requirement = O(n) + O(log n) ≈ O(n)

Trade-off:

While merge sort is not in-place (because of additional space usage), its predictable performance and stability justify the overhead in applications like airline systems, where correctness and speed outweigh minimal extra memory cost.

Conclusion:

Merge Sort provides an efficient, reliable, and scalable approach for managing and retrieving flight schedules in airline booking systems. Its balanced time complexity, stable sorting mechanism, and suitability for large datasets make it one of the most practical algorithms for real-time travel data management.

Design and Analysis of Algorithms (24CS2203)

Merge sort in airline ticket management

T Sai Shreeja 2420030631

G. Divya Sree 2420030287

Course Instructor

Dr. J Sirisha Devi

Professor

Department of Computer Science and Engineering

Case study - statement

This case study focuses on using the Merge Sort algorithm to improve efficiency in airline ticket management systems. Airlines handle large volumes of data daily — including ticket prices, passenger names, and flight schedules — which require fast and accurate sorting for booking, reporting, and display purposes.

The problem arises when traditional sorting methods become slow or unstable as data size increases. Merge Sort, being a divide-and-conquer and stable sorting algorithm with a consistent O(n log n) time complexity, offers a practical solution.

The study demonstrates that implementing Merge Sort allows the system to:

- Efficiently sort large sets of ticket data (by price, name, or date),
- Maintain data order consistency for equal values,
- Improve response time in displaying flight options and passenger records.

In conclusion, the integration of Merge Sort in airline ticket management enhances data organization, system performance, and user experience, making it a reliable choice for handling large-scale airline operations.

Algorithm

1. **Algorithm Steps**
2. **Start**
3. **Input** the list of airline tickets (each containing attributes like Ticket ID, Passenger Name, Price, Departure Time).
4. **If** the list contains only one ticket → return the list (already sorted).
5. **Divide** the list into two halves: Left and Right.
6. **Recursively** apply Merge Sort on both halves.
7. **Merge** the two sorted halves into a single sorted list by comparing:
   Ticket attribute (e.g., price or name) of each element.
8. **Return** the merged sorted list.
9. **Stop**

Pseudo code

```
Algorithm MergeSort(Tickets)
    Input: A list of airline tickets
    Output: Sorted list of tickets

    if length(Tickets) ≤ 1 then
        return Tickets
    end if

    mid ← length(Tickets) / 2
    Left ← Tickets[0 ... mid-1]
    Right ← Tickets[mid ... end]

    Left ← MergeSort(Left)
    Right ← MergeSort(Right)

    return Merge(Left, Right)
end Algorithm
```

```
Algorithm Merge(Left, Right)
    Initialize SortedList as empty
    i ← 0, j ← 0

    while i < length(Left) and j < length(Right) do
        if Left[i].Price ≤ Right[j].Price then
            add Left[i] to SortedList
            i ← i + 1
        else
            add Right[j] to SortedList
            j ← j + 1
        end if
    end while

    Add all remaining elements of Left (if any) to SortedList
    Add all remaining elements of Right (if any) to SortedList

    return SortedList
end Algorithm
```

Time Complexity

Case	Description	Time Complexity
Best Case	Data is already sorted (still divides and merges)	**O(n log n)**
Average Case	Random ticket order	**O(n log n)**
Worst Case	Reverse-sorted or large unsorted data	**O(n log n)**

Space Complexity

Resource	Usage	Complexity
Auxiliary Space	Temporary arrays/lists for merging	**O(n)**
Call Stack (Recursion)	Logarithmic recursion depth	**O(log n)**
Total Space Complexity	Auxiliary + recursion stack	**O(n)**