Report:

Output:

```
Generating 1000000 random orders...
Merge Sort completed in 1.51107 seconds.
std::sort completed in 0.682272 seconds.

First 5 sorted orders (by Merge Sort):
OrderID: 913520, Timestamp: 1700000001, Customer: Customer_913520
OrderID: 147706, Timestamp: 1700000002, Customer: Customer_147706
OrderID: 537173, Timestamp: 1700000003, Customer: Customer_537173
OrderID: 570567, Timestamp: 1700000003, Customer: Customer_570567
OrderID: 488121, Timestamp: 1700000005, Customer: Customer_488121
```

1. Introduction

In large-scale systems such as e-commerce platforms or online services, customer orders need to be processed in chronological order. With datasets reaching up to one million orders, an efficient and reliable sorting algorithm is necessary. This project implements **Merge Sort** to arrange customer orders by timestamp and compares its performance with C++'s built-in std::sort.

---

2. Methodology

1. **Data Structure**:
   Each order is represented as a structure containing orderId, timestamp, and customerName.
2. **Merge Sort Implementation**:
   - The dataset is recursively divided into halves.
   - Each half is sorted independently.
   - The sorted halves are merged into a single sorted list.
     Sorting is based on the **timestamp** field to ensure chronological order.
3. **Dataset Generation**:
   A synthetic dataset of up to **1,000,000 orders** is generated with random timestamps for testing efficiency.
4. **Performance Comparison**:
   - Implemented Merge Sort is compared against std::sort (which uses Introsort: a hybrid of QuickSort, HeapSort, and InsertionSort).
   - Execution times are measured using the C++ <chrono> library.

---

3. Time Complexity Analysis

- **Merge Sort**:
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n log n)
  - Space Complexity: O(n) (extra space for merging)
- **std::sort (Introsort)**:
  - Average Case: O(n log n)
  - Worst Case: O(n log n)
  - Space Complexity: O(log n) (due to recursion depth)
- **Traditional Algorithms**:
  - Bubble Sort, Insertion Sort, Selection Sort → $O(n^2)$, unsuitable for 1M records.
  - Quick Sort → Fast on average but has $O(n^2)$ worst case if not optimized.

---

4. Experimental Results

- **Merge Sort** successfully handled 1 million orders with stable O(n log n) performance.
- **std::sort** was observed to run faster in practice due to hybrid optimizations, though both have the same theoretical complexity.
- Merge Sort guarantees stability (important if two orders share the same timestamp), while std::sort does not guarantee stability in C++ (only in stable_sort).

---

5. Conclusion

- Merge Sort is a **reliable, stable, and scalable algorithm** for arranging large datasets of customer orders by timestamp.
- For practical implementations, std::sort or stable_sort is preferred due to better optimization.
- However, understanding and implementing Merge Sort provides strong insights into divide-and-conquer algorithms and their role in handling **large-scale real-world problems efficiently**.