Exercise 3: Sorting Customer Orders

# 1. Understand Sorting Algorithms

Sorting algorithms are used to arrange data in a specific order to make processing easier and more efficient.

• Bubble Sort: Repeatedly compares adjacent elements and swaps them if they are in the wrong order. Simple but inefficient for large datasets. Time Complexity: O(n^2)

• Insertion Sort: Builds the sorted list one item at a time by inserting items into their correct position. Efficient for small datasets. Time Complexity: O(n^2)

• Quick Sort: A divide-and-conquer algorithm that picks a pivot and partitions the array around the pivot. Fast and efficient. Time Complexity: O(n log n) average, O(n^2) worst.

• Merge Sort: Recursively divides the array into halves, sorts each half, and merges them. Time Complexity: O(n log n)

# 2. Setup

Create a class `Order` with fields: `orderId`, `customerName`, and `totalPrice`.

# 3. Implementation

* Order Class:

public class Order {  
 int orderId;  
 String customerName;  
 double totalPrice;  
  
 public Order(int orderId, String customerName, double totalPrice) {  
 this.orderId = orderId;  
 this.customerName = customerName;  
 this.totalPrice = totalPrice;  
 }  
  
 @Override  
 public String toString() {  
 return orderId + " - " + customerName + " - ₹" + totalPrice;  
 }  
}

* Bubble Sort Implementation:

public class BubbleSort {  
 public static void sort(Order[] orders) {  
 int n = orders.length;  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = 0; j < n - i - 1; j++) {  
 if (orders[j].totalPrice > orders[j + 1].totalPrice) {  
 Order temp = orders[j];  
 orders[j] = orders[j + 1];  
 orders[j + 1] = temp;  
 }  
 }  
 }  
 }  
}

* Quick Sort Implementation:

public class QuickSort {  
 public static void sort(Order[] orders, int low, int high) {  
 if (low < high) {  
 int pi = partition(orders, low, high);  
 sort(orders, low, pi - 1);  
 sort(orders, pi + 1, high);  
 }  
 }  
  
 private static int partition(Order[] orders, int low, int high) {  
 double pivot = orders[high].totalPrice;  
 int i = low - 1;  
 for (int j = low; j < high; j++) {  
 if (orders[j].totalPrice < pivot) {  
 i++;  
 Order temp = orders[i];  
 orders[i] = orders[j];  
 orders[j] = temp;  
 }  
 }  
 Order temp = orders[i + 1];  
 orders[i + 1] = orders[high];  
 orders[high] = temp;  
 return i + 1;  
 }  
}

* Test Class:

public class OrderSortTest {  
 public static void main(String[] args) {  
 Order[] orders = {  
 new Order(101, "Alice", 2000),  
 new Order(102, "Bob", 5000),  
 new Order(103, "Charlie", 3000)  
 };  
  
 BubbleSort.sort(orders);  
 System.out.println("Sorted by Bubble Sort:");  
 for (Order o : orders) System.out.println(o);  
  
 Order[] orders2 = {  
 new Order(101, "Alice", 2000),  
 new Order(102, "Bob", 5000),  
 new Order(103, "Charlie", 3000)  
 };  
  
 QuickSort.sort(orders2, 0, orders2.length - 1);  
 System.out.println("Sorted by Quick Sort:");  
 for (Order o : orders2) System.out.println(o);  
 }  
}

# 4. Analysis

• Bubble Sort: O(n^2) time complexity, inefficient for large lists due to repeated comparisons and swaps.

• Quick Sort: O(n log n) average case, efficient for large data sets due to divide-and-conquer approach.

Quick Sort is generally preferred over Bubble Sort for large inputs due to its faster performance and lower time complexity.

# Output

