**Exercise 3: Sorting Customer Orders**

**OrderSortingSystem.java**

**package** mypackage;

**import** java.util.\*;

**class** Order {

**private** **int** orderId;

**private** String customerName;

**private** **double** totalPrice;

**public** Order(**int** orderId, String customerName, **double** totalPrice) {

**this**.orderId = orderId;

**this**.customerName = customerName;

**this**.totalPrice = totalPrice;

}

**public** **int** getOrderId() { **return** orderId; }

**public** String getCustomerName() { **return** customerName; }

**public** **double** getTotalPrice() { **return** totalPrice; }

@Override

**public** String toString() {

**return** String.*format*("Order ID: %d, Customer: %s, Total Price: ₹%,.2f",

orderId, customerName, totalPrice);

}

}

**public** **class** OrderSortingSystem {

// Bubble Sort implementation

**public** **static** **void** bubbleSort(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].getTotalPrice() < orders[j + 1].getTotalPrice()) {

// Swap

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

// Quick Sort implementation

**public** **static** **void** quickSort(Order[] orders, **int** low, **int** high) {

**if** (low < high) {

**int** pi = *partition*(orders, low, high);

*quickSort*(orders, low, pi - 1);

*quickSort*(orders, pi + 1, high);

}

}

**private** **static** **int** partition(Order[] orders, **int** low, **int** high) {

**double** pivot = orders[high].getTotalPrice();

**int** i = low - 1;

**for** (**int** j = low; j < high; j++) {

**if** (orders[j].getTotalPrice() >= pivot) { // Descending order

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;

}

// Display orders

**public** **static** **void** displayOrders(Order[] orders) {

**for** (Order o : orders) {

System.***out***.println(o);

}

}

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter number of orders: ");

**int** n = sc.nextInt();

sc.nextLine();

Order[] orders = **new** Order[n];

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

System.***out***.println("Enter details for Order " + (i + 1));

System.***out***.print("Order ID: ");

**int** orderId = sc.nextInt();

sc.nextLine();

System.***out***.print("Customer Name: ");

String customerName = sc.nextLine();

System.***out***.print("Total Price: ");

**double** totalPrice = sc.nextDouble();

sc.nextLine();

orders[i] = **new** Order(orderId, customerName, totalPrice);

}

System.***out***.println("\nOrders sorted using Bubble Sort (High to Low Total Price):");

*bubbleSort*(orders);

*displayOrders*(orders);

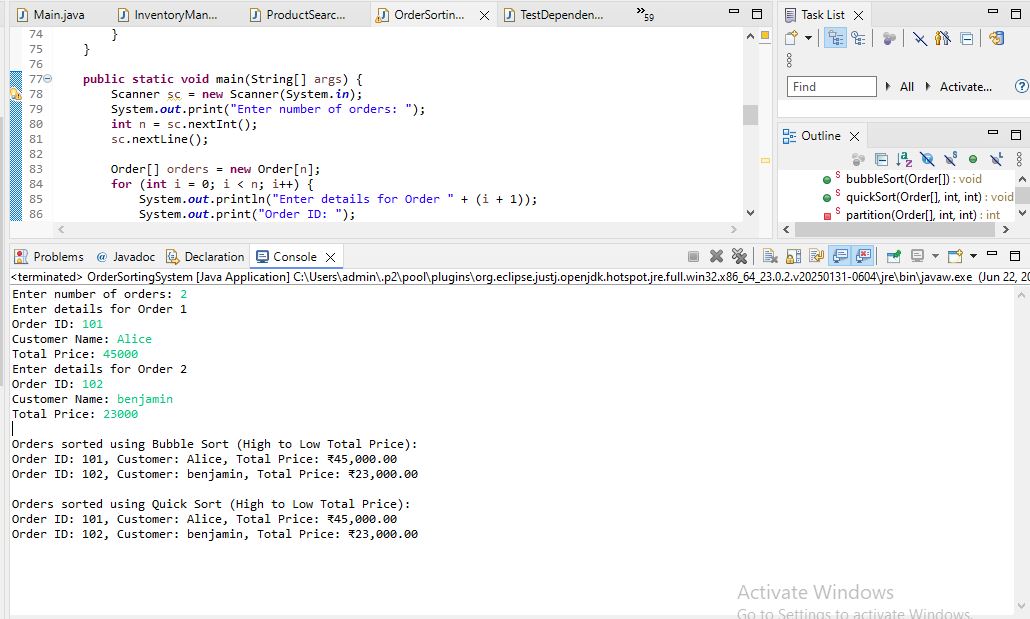
System.***out***.println("\nOrders sorted using Quick Sort (High to Low Total Price):");

*quickSort*(orders, 0, orders.length - 1);

*displayOrders*(orders);

}

}



**1. Understand Sorting Algorithms**

**➔ Bubble Sort:**

* Compares adjacent elements and swaps them if out of order.
* Simple but inefficient for large datasets.
* Time Complexity:
  + Best Case: O(n) (if already sorted)
  + Average Case: O(n²)
  + Worst Case: O(n²)

**➔ Insertion Sort:**

* Builds the sorted array one element at a time.
* Good for small datasets.
* Time Complexity:
  + Best Case: O(n)
  + Average & Worst Case: O(n²)

**➔ Quick Sort:**

* Divide and conquer algorithm.
* Picks a pivot, partitions the array, recursively sorts partitions.
* Very efficient for large datasets.
* Time Complexity:
  + Best & Average Case: O(n log n)
  + Worst Case: O(n²) (if poorly chosen pivot)

**➔ Merge Sort:**

* Also divide and conquer.
* Always divides into halves, merges back sorted.
* Stable, guaranteed O(n log n) even in worst case.
* Requires extra space.

**2. Setup**

* Class Order is created with:
  + orderId
  + customerName
  + totalPrice

**3. Implementation**

* **Bubble Sort** and **Quick Sort** are implemented to sort orders by totalPrice (descending).
* Input is taken from user for orders.
* Both sorting methods are applied and displayed.

**4. Analysis**

**➔ Performance Comparison**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |

* Bubble Sort performs poorly for large data.
* Quick Sort is much faster for most cases.

**➔ Why is Quick Sort preferred?**

* Much faster for large data.
* Average time complexity O(n log n).
* Bubble Sort’s quadratic time makes it unsuitable for real-world use.