# Exercise 3: Sorting Customer Orders

***Sorting Algorithms***

| **Algorithm** | **Best Case** | **Avg Case** | **Worst Case** | **Space** | **Stable** | **How It Works** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) | Yes | Repeatedly compares adjacent elements and swaps them if they are in the wrong order. Largest elements "bubble up" to the end. | Small datasets, educational purposes |
| Insertion Sort | O(n) | O(n²) | O(n²) | O(1) | Yes | Builds a sorted array one element at a time by inserting each new element into its correct position in the already sorted part. | Small/nearly sorted data, streaming inputs |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) | No | Uses a pivot to partition the array into two subarrays (elements < pivot and > pivot), then recursively sorts each subarray. | General-purpose sorting, large datasets |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) | Yes | Divides the array into halves, recursively sorts each half, then merges them back in order. | Large datasets, stable sorting, linked lists |

***Implementation***

Order.java:

package com.example;

public class Order {

private String orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

*@Override*

public String toString() {

return "Order [ID=" + orderId + ", Customer=" + customerName

+ ", Total=Rs " + totalPrice + "]";

}

}

OrderSorter.java:

package com.example;

public class OrderSorter {

// Bubble Sort

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()) {

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

// Quick Sort

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

if (low < high) {

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

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

*quickSort*(orders, pivotIndex + 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) {

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;

}

}

Main.java:

package com.example;

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order("O101", "Alice", 150.0),

new Order("O102", "Bob", 75.5),

new Order("O103", "Charlie", 210.0),

new Order("O104", "Diana", 60.0)

};

System.***out***.println("Original Orders:");

*printOrders*(orders);

// Bubble Sort

OrderSorter.*bubbleSort*(orders);

System.***out***.println("\nAfter Bubble Sort:");

*printOrders*(orders);

// Quick Sort

Order[] orders2 = {

new Order("O101", "Alice", 150.0),

new Order("O102", "Bob", 75.5),

new Order("O103", "Charlie", 210.0),

new Order("O104", "Diana", 60.0)

};

OrderSorter.*quickSort*(orders2, 0, orders2.length - 1);

System.***out***.println("\nAfter Quick Sort:");

*printOrders*(orders2);

}

private static void printOrders(Order[] orders) {

for (Order order : orders) {

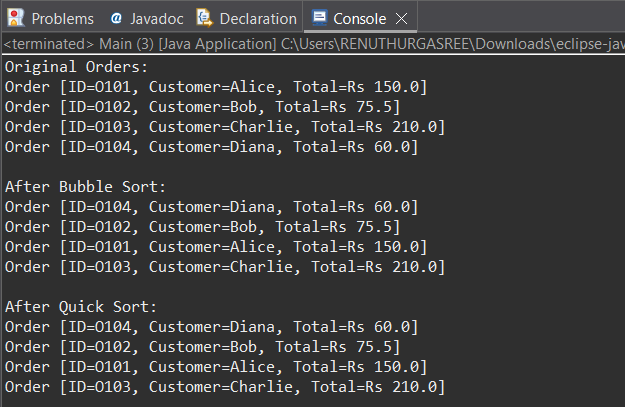
System.***out***.println(order);

}

}

}

Output:



***Analysis***

Bubble sort vs Quick sort:

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **In-Place?** | **Stable?** |
| --- | --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | Yes | Yes |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | Yes | No |

Why quick sort is preferred ?

* It has faster average case which is better for large datasets.
* No extra space needed as it is an in-place sorting algorithm.
* It works well with CPU caches.