**Exercise 3: Sorting Customer Orders**

Scenario: Tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

1. Understand Sorting Algorithms: Sorting algorithms are essential for organizing data to allow fast access, comparison, or processing. Different sorting algorithms have varying time complexities:

* **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order. Time Complexity: O(n^2)
* **Insertion Sort:** Builds the final sorted array one item at a time. Time Complexity: O(n^2)
* **Merge Sort:** A divide-and-conquer algorithm that splits and merges arrays. Time Complexity: O(n log n)
* **Quick Sort:** Picks a pivot and partitions the array. Time Complexity: Average O(n log n), Worst O(n^2)

1. Setup: Create a class Order with the following attributes:

* String orderId
* String customerName
* double totalPrice

1. Implementation:

Java Code:

class Order {

String orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

return orderId + " | " + customerName + " | ₹" + totalPrice;

}

}

class SortOrders {

public static void bubbleSort(Order[] orders) {

for (int i = 0; i < orders.length - 1; i++) {

for (int j = 0; j < orders.length - i - 1; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

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);

}

}

public 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;

}

public static void main(String[] args) {

Order[] orders = {

new Order("O101", "Alice", 450.00),

new Order("O102", "Bob", 199.99),

new Order("O103", "Charlie", 999.00)

};

System.out.println("Before Sorting:");

for (Order o : orders) System.out.println(o);

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

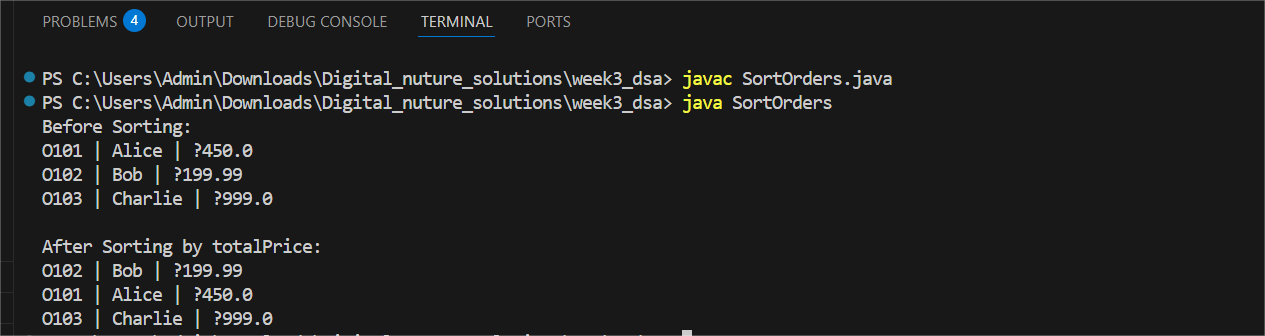
System.out.println("\nAfter Sorting by totalPrice:");

for (Order o : orders) System.out.println(o);

}

}

OUTPUT:



1. Analysis:

* **Bubble Sort:** O(n^2). Simple but inefficient for large data sets.
* **Quick Sort:** O(n log n) average, O(n^2) worst case. More efficient due to its divide-and-conquer approach.

Quick Sort is preferred in real-world scenarios due to its speed and lower space usage compared to Merge Sort. Use Bubble Sort only for educational or very small inputs.