**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**1. Understanding Sorting Algorithms**

**Overview of Sorting Algorithms**

* **Bubble Sort:** A basic comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if necessary.
  + **Best-Case:** O(n) (when the array is already sorted).
  + **Average and Worst-Case:** O(n²).
* **Insertion Sort:** Builds a sorted array one element at a time by inserting each new element into its correct position in the already sorted portion.
  + **Best-Case:** O(n) (when the array is already sorted).
  + **Average and Worst-Case:** O(n²).
* **Quick Sort:** A divide-and-conquer algorithm that partitions the array based on a pivot element and recursively sorts the sub-arrays.
  + **Best and Average-Case:** O(n log n).
  + **Worst-Case:** O(n²) (with poor pivot selection).
* **Merge Sort:** A divide-and-conquer algorithm that splits the array into halves, sorts each half, and merges the sorted halves.
  + **Best, Average, and Worst-Case:** O(n log n).
* **2. Setup**

Create a Class Order with Attributes

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 "Order ID: " + orderId + ", Customer: " + customerName + ", Total Price: $" + totalPrice;

}

}

**3. Implementation**

import java.util.Arrays;

import java.util.Comparator;

public class SortingCustomerOrders {

// 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].totalPrice > orders[j + 1].totalPrice) {

// Swap orders[j] and orders[j + 1]

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 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].totalPrice;

int i = low - 1;

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

if (orders[j].totalPrice <= pivot) {

i++;

// Swap orders[i] and orders[j]

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// Swap orders[i + 1] and orders[high]

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

public static void main(String[] args) {

// Creating an array of orders

Order[] orders = {

new Order(1, "Alice", 250.00),

new Order(2, "Bob", 120.00),

new Order(3, "Charlie", 300.00),

new Order(4, "Diana", 150.00),

new Order(5, "Eve", 180.00)

};

// Display unsorted orders

System.out.println("Unsorted Orders:");

for (Order order : orders) {

System.out.println(order);

}

// Sort orders using Bubble Sort

Order[] bubbleSortedOrders = Arrays.copyOf(orders, orders.length);

bubbleSort(bubbleSortedOrders);

System.out.println("\nOrders Sorted by Bubble Sort:");

for (Order order : bubbleSortedOrders) {

System.out.println(order);

}

// Sort orders using Quick Sort

Order[] quickSortedOrders = Arrays.copyOf(orders, orders.length);

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

System.out.println("\nOrders Sorted by Quick Sort:");

for (Order order : quickSortedOrders) {

System.out.println(order);

}

}

}

**4. Analysis**

**Compare the Performance (Time Complexity)**

* **Bubble Sort**:
  + **Time Complexity**: O(n²) in the worst and average cases.
  + **Best-Case**: O(n) (when the array is already sorted).
* **Quick Sort**:
  + **Time Complexity**: O(n log n) on average.
  + **Worst-Case**: O(n²) (with poor pivot choices).

**Why Quick Sort is Generally Preferred**:

* **Efficiency**: Quick Sort is more efficient than Bubble Sort for large datasets due to its O(n log n) average time complexity.
* **Performance**: Quick Sort performs well in practice with good pivot selection strategies and can be optimized to handle worst-case scenarios better