**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. Understand Sorting Algorithms**

**Bubble Sort:**  
A simple comparison-based algorithm. Repeatedly steps through the list, compares adjacent items, and swaps them if they are in the wrong order.

* Time Complexity: O(n^2)
* Space Complexity: O(1)

**Insertion Sort:**  
Builds the final sorted array one item at a time. It’s efficient for small data sets.

* Time Complexity: O(n^2)
* Space Complexity: O(1)

**Quick Sort:**  
A divide-and-conquer algorithm. Picks a pivot, partitions the array, and recursively sorts the partitions.

* Average Time Complexity: O(n log n)
* Worst Time Complexity: O(n^2) (rare with good pivot choice)
* Space Complexity: O(log n)

**Merge Sort:**  
Divides the list into halves, sorts each recursively, and then merges them.

* Time Complexity: O(n log n)
* Space Complexity: O(n)

**2. Setup**

**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 "Order [ID=" + orderId + ", Name=" + customerName + ", Price=" + totalPrice + "]";

}

}

**3. Implementation**

**Sorting Class:**

public class Sorting {

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) {

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);

}

}

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;

}

}

**Main Class:**

import java.util.\*;

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 550.50),

new Order(2, "Bob", 320.00),

new Order(3, "Charlie", 790.25),

new Order(4, "David", 130.00)

};

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

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

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

Sorting.bubbleSort(orders);

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

Order[] orders2 = {

new Order(1, "Alice", 550.50),

new Order(2, "Bob", 320.00),

new Order(3, "Charlie", 790.25),

new Order(4, "David", 130.00)

};

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

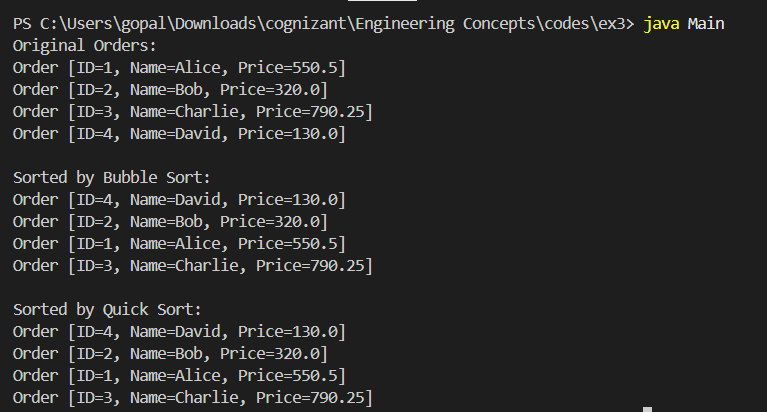
Sorting.quickSort(orders2, 0, orders2.length - 1);

for (Order o : orders2) System.out.println(o);

}

}

**Output:**



**4. Analysis**

**Bubble Sort:**

* Worst Time Complexity: O(n^2)
* It is Inefficient for large datasets
* Simple but slow

**Quick Sort:**

* Average Time Complexity: O(n log n)
* Much faster for large datasets
* It is mostly preferred for sorting large sets of orders

**Conclusion:**  
Quick Sort is generally preferred over Bubble Sort due to its significantly better average-case performance, making it suitable for real-world applications where order sorting speed is essential.