**Exercise – 3 Sorting Customer Orders**

**Bubble Sort**

Bubble Sort is a simple comparison-based algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order.

Best Case: O(n) → when the array is already sorted.

Average/Worst Case: O(n²) → slow for large datasets.

**Insertion Sort**

Insertion Sort builds the final sorted array one item at a time, by inserting each element into its correct position.

Best Case: O(n) → when the data is nearly sorted.

Average/Worst Case: O(n²) → inefficient for large, unsorted data.

**Quick Sort**

Quick Sort is a divide-and-conquer algorithm. It:

* Picks a pivot.
* Partitions elements into less than and greater than the pivot.
* Recursively sorts the partitions.

Best/Average Case: O(n log n)

Worst Case: O(n²) → if pivot is poorly chosen (e.g., already sorted array without random pivoting)

**Merge Sort**

Merge Sort divides the array into halves, recursively sorts them, and then merges the sorted halves.

Time Complexity (All Cases): O(n log n)

Preferred when stable sort and consistent performance is needed.

**Comparison of performance (time complexity) of Bubble Sort and Quick Sort**

|  |  |  |
| --- | --- | --- |
| **Case** | **Bubble Sort** | **Quick Sort** |
| Best Case | O(n) | O(n logn) |
| Average Case | O(n^2) | O(n logn) |
| Worst Case | O(n^2) | O(n^2) |

**Quick Sort** is generally preferred because it offers much better performance, especially for large and unsorted datasets. **Bubble Sort**, while easy to understand, is inefficient and not suitable for practical use.

**Main.java**

public class Main {

public static void main(String[] args) {

Order[] orders1 = {

new Order(101, "Alice", 2000),

new Order(102, "Bob", 5000),

new Order(105, "Charlie", 4000),

new Order(104, "David", 3000),

new Order(103, "Eva", 6000)

};

Order[] orders2 = {

new Order(101, "Alice", 2000),

new Order(102, "Bob", 7000),

new Order(105, "Eva", 4000),

new Order(104, "David", 3000),

new Order(103, "Charlie", 6000)

};

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

displayOrders(orders1);

bubbleSort(orders1);

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

displayOrders(orders1);

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

displayOrders(orders2);

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

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

displayOrders(orders2);

}

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;

}

}

}

}

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);

}

}

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

Order pivot = orders[high];

int i = low - 1;

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

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

i++;

swap(orders, i, j);

}

}

swap(orders, i + 1, high);

return i + 1;

}

static void swap(Order[] orders, int i, int j) {

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

static void displayOrders(Order[] orders) {

for (Order o : orders) {

System.out.println("OrderID: " + o.orderId + ", Name: " + o.customerName + ", Price: " + o.totalPrice);

}

}

}

**Order.java**

class Order {

int orderId;

String customerName;

int totalPrice;

Order(int orderId, String customerName, int totalPrice) {

this.orderId = orderId;

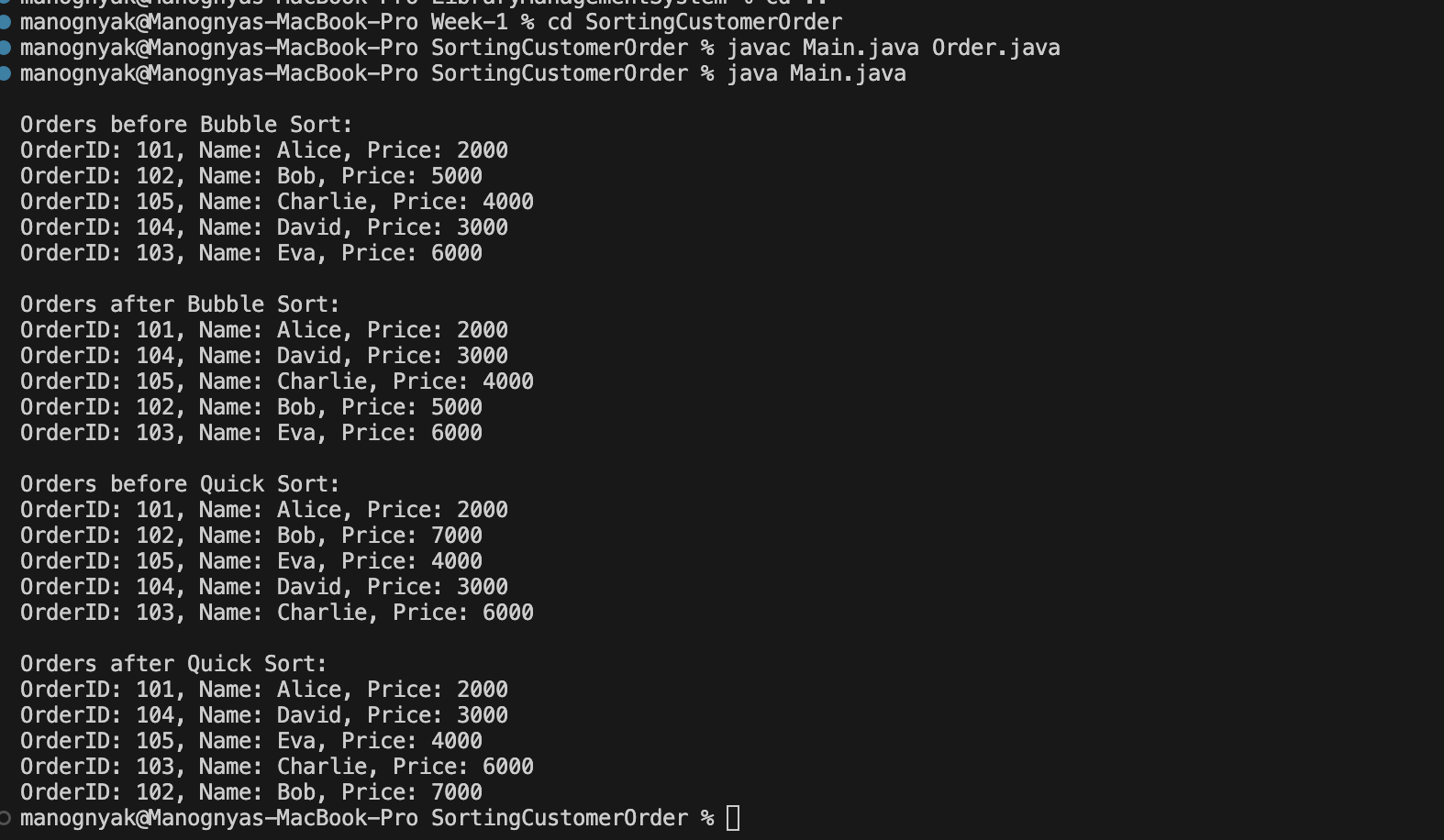
this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

**Output**

****