# Exercise 3: Sorting Customer Orders

### 

### **Understand Sorting Algorithms**

**Bubble Sort:**

* **Description:** A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.
* **Time Complexity:**
  + Best Case: O(n)O(n)O(n) (when the list is already sorted)
  + Average Case: O(n2)O(n^2)O(n2)
  + Worst Case: O(n2)O(n^2)O(n2)

**Insertion Sort:**

* **Description:** Builds the final sorted array one item at a time, taking each element and inserting it into its correct position among the already sorted elements.
* **Time Complexity:**
  + Best Case: O(n)O(n)O(n) (when the list is already sorted)
  + Average Case: O(n2)O(n^2)O(n2)
  + Worst Case: O(n2)O(n^2)O(n2)

**Quick Sort:**

* **Description:** A divide-and-conquer algorithm that picks an element as a pivot and partitions the array around the pivot. The process is then repeated recursively for the sub-arrays.
* **Time Complexity:**
  + Best Case: O(nlog⁡n)O(n \log n)O(nlogn)
  + Average Case: O(nlog⁡n)O(n \log n)O(nlogn)
  + Worst Case: O(n2)O(n^2)O(n2) (when the pivot selection is poor, e.g., always picking the smallest or largest element)

**Merge Sort:**

* **Description:** A divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and then merges the two sorted halves.
* **Time Complexity:**
  + Best Case: O(nlog⁡n)O(n \log n)O(nlogn)
  + Average Case: O(nlog⁡n)O(n \log n)O(nlogn)
  + Worst Case: O(nlog⁡n)O(n \log n)O(nlogn)

### **Step 2: Setup**

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

// Getters

public int getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

}

### **Step 3: Implementation**

**Bubble Sort:**

public class BubbleSort {

public static void bubbleSort(Order[] orders) {

int n = orders.length;

boolean swapped;

for (int i = 0; i < n - 1; i++) {

swapped = false;

for (int j = 0; j < n - i - 1; j++) {

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

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

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = true;

}

}

// If no two elements were swapped in the inner loop, then break

if (!swapped) break;

}

}

}

**Quick Sort:**

public class QuickSort {

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].getTotalPrice();

int i = low - 1;

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

if (orders[j].getTotalPrice() <= 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] (or pivot)

Order temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

}

### **Step 4: Analysis**

**Time Complexity:**

* **Bubble Sort:**
  + Best Case: O(n)O(n)O(n) (when the array is already sorted)
  + Average Case: O(n2)O(n^2)O(n2)
  + Worst Case: O(n2)O(n^2)O(n2)
* **Quick Sort:**
  + Best Case: O(nlog⁡n)O(n \log n)O(nlogn)
  + Average Case: O(nlog⁡n)O(n \log n)O(nlogn)
  + Worst Case: O(n2)O(n^2)O(n2) (when the pivot selection is poor)

**Comparison and Suitability:**

* **Bubble Sort:**
  + Easy to implement but inefficient for large datasets.
  + Suitable only for small datasets or when simplicity is more important than performance.
* **Quick Sort:**
  + Efficient and performs well on average for large datasets.
  + More complex to implement but generally preferred for its better average-case performance.

**Conclusion:** For sorting customer orders by total price on an e-commerce platform, **Quick Sort** is generally preferred due to its efficient average-case performance of O(nlog⁡n)O(n \log n)O(nlogn). It handles large datasets much better than Bubble Sort, which has a quadratic time complexity in the average and worst cases.