## Exercise 3: Sorting Customer Orders

### Step 1: Understand Sorting Algorithms

#### Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).

#### Bubble Sort

* **Description**: Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.
* **Time Complexity**:
  + Best Case: O(n)
  + Average Case: O(n^2)
  + Worst Case: O(n^2)
* **Space Complexity**: O(1) (in-place sorting)

#### Insertion Sort

* **Description**: Insertion Sort builds the sorted array one item at a time. It takes each element and inserts it into its correct position in the sorted part of the array.
* **Time Complexity**:
  + Best Case: O(n)
  + Average Case: O(n^2)
  + Worst Case: O(n^2)
* **Space Complexity**: O(1) (in-place sorting)

#### Quick Sort

* **Description**: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two sub-arrays, according to whether elements are less than or greater than the pivot. The sub-arrays are then sorted recursively.
* **Time Complexity**:
  + Best Case: O(n log n)
  + Average Case: O(n log n)
  + Worst Case: O(n^2) (rare and depends on pivot selection)
* **Space Complexity**: O(log n) (in-place sorting, but recursive)

#### Merge Sort

* **Description**: Merge Sort is a divide-and-conquer algorithm that divides the array into halves, sorts each half, and then merges the sorted halves to produce the final sorted array.
* **Time Complexity**:
  + Best Case: O(n log n)
  + Average Case: O(n log n)
  + Worst Case: O(n log n)
* **Space Complexity**: O(n) (not in-place)

### Step 2: Setup

#### Define the Order Class

// Java implementation

public class Order {

private String orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

// Getters

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

}

### Step 3: Implementation

#### Bubble Sort Algorithm

// Java implementation of Bubble Sort

public class BubbleSort {

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].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;

}

}

}

}

}

#### Quick Sort Algorithm

// Java implementation of Quick Sort

public class QuickSort {

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

// Recursively sort elements before partition and after partition

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); // Index of smaller element

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

// If current element is smaller than or equal to pivot

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 Comparison

* **Bubble Sort**:
  + Best Case: O(n) (when the array is already sorted)
  + Average Case: O(n^2)
  + Worst Case: O(n^2)
* **Quick Sort**:
  + Best Case: O(n log n) (when the pivot divides the array into two nearly equal halves)
  + Average Case: O(n log n)
  + Worst Case: O(n^2) (when the pivot is the smallest or largest element)

#### Preference of Quick Sort over Bubble Sort

* **Efficiency**: Quick Sort is generally faster than Bubble Sort due to its average and best-case time complexities of O(n log n) compared to Bubble Sort's O(n^2).
* **Scalability**: Quick Sort handles larger datasets more efficiently.

**Practical Use**: Although Quick Sort has a worst-case time complexity of O(n^2), it can be mitigated by using good pivot selection strategies (like median-of-three) or randomized pivot selection.