# Exercise 3: Sorting Customer Orders

## 1. Understand Sorting Algorithms

### **Bubble Sort:**

- Description: Continuously steps through the list, comparing adjacent elements and swapping them if they are out of order. This process repeats until the entire list is sorted.

- Time Complexity:

- Best Case: O(n) when the list is already sorted, and only one pass is needed.

- Average Case: O(n²) due to the need for multiple passes.

- Worst Case: O(n²) when the list is sorted in reverse order.

- Space Complexity: O(1), as it sorts in place.

- Characteristics: Simple to implement but inefficient for large datasets due to its quadratic time complexity.

### **Insertion Sort:**

- Description: Builds the sorted array one item at a time. It takes each element and inserts it into its correct position in the already sorted part of the array.

- Time Complexity:

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

- Average Case: O(n²) due to repeated comparisons and shifts.

- Worst Case: O(n²) when the array is sorted in reverse order.

- Space Complexity: O(1), as it sorts in place.

- Characteristics: More efficient than Bubble Sort for small or nearly sorted datasets.

### **Quick Sort:**

- Description: A divide-and-conquer algorithm that selects a 'pivot' element, partitions the array into two sub-arrays (elements less than the pivot and elements greater than the pivot), and recursively sorts the sub-arrays.

- Time Complexity:

- Best Case: O(n log n) when the pivot divides the array into nearly equal halves.

- Average Case: O(n log n).

- Worst Case: O(n²) when the pivot is consistently the smallest or largest element, though this can be mitigated with good pivot selection strategies.

- Space Complexity: O(log n) due to the recursion stack.

- Characteristics: Very efficient for large datasets and commonly used due to its average-case efficiency.

### **Merge Sort:**

- Description: A divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and then merges the sorted halves.

- Time Complexity:

- Best Case: O(n log n).

- Average Case: O(n log n).

- Worst Case: O(n log n).

- Space Complexity: O(n) due to the need for temporary arrays during the merge process.

- Characteristics: Stable and consistent in performance, but uses more memory compared to Quick Sort.

**2. Setup**

Create a Class `Order`:

- Purpose: Represents a customer order with attributes such as order ID, customer name, and total price.

- Methods:

- Constructor: Initializes an `Order` instance with specified attributes.

- Getters: Retrieve the values of `orderId`, `customerName`, and `totalPrice`.

- Setters: Modify the values of `orderId`, `customerName`, and `totalPrice`.

**3. Implementation**

**Bubble Sort Implementation:**

- Method `bubbleSort(Order[] orders)`

- Purpose: Sorts an array of `Order` objects by total price using the Bubble Sort algorithm.

- Functionality: Repeatedly compares adjacent orders and swaps them if they are out of order. Continues until no more swaps are needed, indicating the array is sorted.

**Quick Sort Implementation:**

- Method `quickSort(Order[] orders, int low, int high)`

- Purpose: Sorts an array of `Order` objects by total price using the Quick Sort algorithm.

- Functionality: Recursively sorts the array by selecting a pivot, partitioning the array into elements less than and greater than the pivot, and then sorting each partition.

**- Method `partition(Order[] orders, int low, int high)`**

- Purpose: Partitions the array around a pivot for use in Quick Sort.

- Functionality: Rearranges the elements so that those less than the pivot come before it and those greater come after it, then returns the index of the pivot.

**4. Analysis**

**Time Complexity Comparison:**

**- Bubble Sort:**

- Best Case: O(n) (already sorted array).

- Average Case: O(n²).

- Worst Case: O(n²) (reverse-sorted array).

- Space Complexity: O(1).

**- Quick Sort:**

- Best Case: O(n log n) (balanced pivot selection).

- Average Case: O(n log n).

- Worst Case: O(n²) (poor pivot selection).

- Space Complexity: O(log n).

**Why Quick Sort is Generally Preferred:**

**1. Efficiency:** Quick Sort typically performs faster than Bubble Sort due to its O(n log n) time complexity compared to Bubble Sort's O(n²).

**2. Scalability:** Handles large datasets more effectively, providing better performance on average.

**3. Memory Usage:** Uses less additional memory compared to Merge Sort, making it more suitable for large arrays.