**Exercise 3: Sorting Customer Orders**

## Step 1: Understand Sorting Algorithms

### Bubble Sort

Bubble Sort is 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)
* Average Case: O(n^2)
* Worst Case: O(n^2)

### Insertion Sort

Insertion Sort builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

**Time Complexity:**

* Best Case: O(n)
* Average Case: O(n^2)
* Worst Case: O(n^2)

### Quick Sort

Quick Sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

**Time Complexity:**

* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n^2) (rare, and can be mitigated with good pivot selection)

### Merge Sort

Merge Sort is an efficient, stable, comparison-based, divide and conquer sorting algorithm. Most implementations produce a stable sort, meaning that the implementation preserves the input order of equal elements in the sorted output.

**Time Complexity:**

* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n log n)

## Step 2: Setup

We will create a Java class Order with attributes orderId, customerName, and totalPrice.

**Code:**

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{" +

"orderId=" + orderId +

", customerName='" + customerName + '\'' +

", totalPrice=" + totalPrice +

'}';

}

}

## Step 3: Implementation

### Bubble Sort Implementation in Java:

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].totalPrice > orders[j+1].totalPrice) {

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

Order temp = orders[j];

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

orders[j+1] = temp;

}

}

}

}

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Customer1", 100.0),

new Order(2, "Customer2", 50.0),

new Order(3, "Customer3", 200.0)

};

System.out.println("Before Sorting:");

for (Order order : orders) {

System.out.println(order);

}

bubbleSort(orders);

System.out.println("\nAfter Sorting:");

for (Order order : orders) {

System.out.println(order);

}

}

}

**Quick Sort Implementation in Java:**

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);

}

}

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

double pivot = orders[high].totalPrice;

int i = (low-1); // index of smaller element

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

if (orders[j].totalPrice <= 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;

}

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Customer1", 100.0),

new Order(2, "Customer2", 50.0),

new Order(3, "Customer3", 200.0)

};

System.out.println("Before Sorting:");

for (Order order : orders) {

System.out.println(order);

}

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

System.out.println("\nAfter Sorting:");

for (Order order : orders) {

System.out.println(order);

}

}

}

## Step 4: Analysis

### Performance Comparison

**Bubble Sort:**

* Time Complexity: O(n^2) in the average and worst cases.
* Inefficient for large datasets.

**Quick Sort:**

* Time Complexity: O(n log n) on average.
* More efficient than Bubble Sort for larger datasets.
* Worst-case complexity is O(n^2), but this can be mitigated with good pivot selection strategies (e.g., median-of-three).

### Why Quick Sort is Generally Preferred Over Bubble Sort

* **Efficiency**: Quick Sort has a better average-case time complexity (O(n log n)) compared to Bubble Sort (O(n^2)), making it much faster for large datasets.
* **Scalability**: Quick Sort is more scalable and performs well on a larger set of data.
* **Optimization**: There are many optimizations available for Quick Sort (e.g., choosing a good pivot) that make it even more efficient in practice.