**Name : Madhavakumar D**

**Registered Email ID : madhavakumar.d@gmail.com**

**Superset ID : 5018705**

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Solution:**

1. **Understanding:**

#### 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 process is repeated until the list is sorted.

* **Time Complexity**:
  + Best Case: O(n) (when the array is already sorted)
  + Average Case: O(n²)
  + Worst Case: O(n²)
* **Space Complexity**: O(1)

#### 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) (when the array is already sorted)
  + Average Case: O(n²)
  + Worst Case: O(n²)
* **Space Complexity**: O(1)

#### Quick Sort

Quick Sort is a highly efficient sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

* **Time Complexity**:
  + Best Case: O(n log n)
  + Average Case: O(n log n)
  + Worst Case: O(n²) (when the pivot selection is poor, e.g., always picking the largest or smallest element)
* **Space Complexity**: O(log n) due to the recursion stack

#### 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**: O(n log n) for all cases
* **Space Complexity**: O(n)

1. **Setup and 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;

}

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

@Override

public String toString() {

return "Order ID: " + orderId + ", Customer Name: " + customerName + ", Total Price: " + totalPrice;

}

}

public class Sorting {

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;

}

}

}

}

}

public class Sorting {

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

if (low < high) {

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

quickSort(orders, low, pi - 1); // Recursively sort elements before partition

quickSort(orders, pi + 1, high); // Recursively sort elements after partition

}

}

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;

}

}

1. **Analysis:**

#### Performance Comparison

**Bubble Sort**:

* **Best Case**: O(n) (when the array is already sorted)
* **Average Case**: O(n²)
* **Worst Case**: O(n²)

**Quick Sort**:

* **Best Case**: O(n log n)
* **Average Case**: O(n log n)
* **Worst Case**: O(n²) (when the pivot selection is poor)

#### Why Quick Sort is Preferred?

* **Efficiency**: Quick Sort is generally faster than Bubble Sort, especially for larger datasets, due to its O(n log n) average time complexity.
* **Practical Performance**: In practice, Quick Sort's cache performance and low-overhead recursive implementation make it a good choice for many applications.
* **Flexibility**: Quick Sort can be easily adapted for various pivot selection strategies and optimizations.