**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.

import java.util.Arrays;

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

public class OrderSorting {

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) {

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

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].totalPrice;

int i = (low - 1);

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

if (orders[j].totalPrice <= pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

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, "Yash", 100),

new Order(2, "Bruno", 250),

new Order(3, "Neetu", 50),

new Order(4, "Harya", 150)

};

System.***out***.println("Before sorting:");

for (Order order : orders) {

System.***out***.println(order.orderId + ", " + order.customerName + ", " + order.totalPrice);

}

*bubbleSort*(orders);

System.***out***.println("\nAfter Bubble Sort:");

for (Order order : orders) {

System.***out***.println(order.orderId + ", " + order.customerName + ", " + order.totalPrice);

}

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

System.***out***.println("\nAfter Quick Sort:");

for (Order order : orders) {

System.***out***.println(order.orderId + ", " + order.customerName + ", " + order.totalPrice);

}

}

}

OUTPUT:

3, Neetu, 50.0

1, Yash, 100.0

4, Harya, 150.0

2, Bruno, 250.0

After Quick Sort:

3, Neetu, 50.0

1, Yash, 100.0

4, Harya, 150.0

2, Bruno, 250.0

1)Bubble Sort

Compares adjacent elements and swaps them if they are in the wrong order.

Repeatedly passes through the list until no swaps occur.

Inefficient for large datasets.

Time complexity: O(n^2).

2) Insertion Sort

Builds the sorted array one element at a time.

Compares the current element with the sorted portion and inserts it in the correct position.

Efficient for small datasets or partially sorted data.

Time complexity: O(n^2).

3)Quick Sort

A divide-and-conquer algorithm.

Picks a pivot element and partitions the array around it.

Recursively sorts the sub-arrays.

Generally efficient for most datasets.

Time complexity: O(n log n) on average.

4)Merge Sort

A divide-and-conquer algorithm.

Divides the array into halves, recursively sorts them, and merges the sorted halves.

Efficient for large datasets.

Time complexity: O(n log n).

5)Analysis of Time Complexity:

Bubble Sort: O(n^2)

Quick Sort: O(n log n) on average

6)Why Quick Sort is Preferred:

Quick Sort is generally faster than Bubble Sort for larger datasets due to its average-case time complexity of O(n log n) compared to O(n^2) for Bubble Sort. Quick Sort also has better cache performance and is more efficient in practice.