**WEEK 1 - ALGORITHMS AND DATA STRUCTUERS**

**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.

**SORTING ALGORITHMS:**

### ****BUBBLE SORT****

Compares and swaps adjacent elements repeatedly until the list is sorted.  
Time Complexity: Best – O(n), Average/Worst – O(n²)

### ****INSERTION SORT****

Builds the sorted list one item at a time by inserting elements into their correct position.  
Time Complexity: Best – O(n), Average/Worst – O(n²)

### ****QUICK SORT****

Divides the list using a pivot and recursively sorts the partitions (divide and conquer).  
Time Complexity: Best/Average – O(n log n), Worst – O(n²)

### ****MERGE SORT****

Recursively splits the list into halves, sorts them, and merges back into a single sorted list.  
Time Complexity: Best/Average/Worst – O(n log n).

**SETUP AND IMPLEMENTATION:**

### Order.java

public class Order {

int id;

String name;

double price;

public Order(int id, String name, double price) {

this.id = id; this.name = name; this.price = price;

}

public String toString() {

return id + " | " + name + " | ₹" + price;

}

}

### OrderManager.java

public class OrderManager {

void bubbleSort(Order[] o) {

for (int i = 0; i < o.length - 1; i++)

for (int j = 0; j < o.length - i - 1; j++)

if (o[j].price > o[j + 1].price) {

Order temp = o[j]; o[j] = o[j + 1]; o[j + 1] = temp;

}

System.out.println("\nBubble Sort:");

show(o);

}

void quickSort(Order[] o, int low, int high) {

if (low < high) {

int pi = part(o, low, high);

quickSort(o, low, pi - 1);

quickSort(o, pi + 1, high);

}

}

int part(Order[] o, int low, int high) {

double pivot = o[high].price;

int i = low - 1;

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

if (o[j].price < pivot) {

i++; Order t = o[i]; o[i] = o[j]; o[j] = t;

}

Order t = o[i + 1]; o[i + 1] = o[high]; o[high] = t;

return i + 1;

}

void show(Order[] o) {

for (Order a : o) System.out.println(a);

}

}

### Main.java

import java.util.;

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Arjun", 5000),

new Order(2, "Sneha", 2500),

new Order(3, "Ravi", 1500)

};

OrderManager m = new OrderManager();

m.bubbleSort(orders.clone());

System.out.println("\nQuick Sort:");

Order[] qSorted = orders.clone();

m.quickSort(qSorted, 0, qSorted.length - 1);

m.show(qSorted);

}

}

### OUTPUT:

Bubble Sort:

3 | Ravi | ₹1500.0

2 | Sneha | ₹2500.0

1 | Arjun | ₹5000.0

Quick Sort:

3 | Ravi | ₹1500.0

2 | Sneha | ₹2500.0

1 | Arjun | ₹5000.0

**TIME COMPLEXITY:**

**Bubble Sort**

This works by repeatedly swapping adjacent elements if they are in the wrong order. It continues until no more swaps are needed. Although easy to implement, it is inefficient for large datasets.

* **Time Complexity**:
  + **Best Case**: O(n) *(already sorted)*
  + **Average & Worst Case**: O(n²)

**Quick Sort**

This works under **divide-and-conquer** algorithm. It picks a pivot element, partitions the array, and recursively sorts the partitions. It is much more efficient in practice.

* **Time Complexity**:
  + **Best & Average Case**: O(n log n)
  + **Worst Case**: O(n²) *(rare, when pivot selection is poor)*

**REASONS FOR CHOOSING QUICK SORT**

Quick Sort is generally preferred over Bubble Sort because:

* It is **much faster** in real-world use cases due to its efficient partitioning.
* It uses **less memory** compared to Merge Sort and is more scalable than Bubble Sort.
* Even for moderately sized lists like customer orders, Quick Sort handles sorting more effectively with better performance.