**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.

1. Understand Search Algorithms:

* + Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
* **Bubble Sort**: **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
* **Algorithm:**
* *traverse from left and compare adjacent elements and the higher one is placed at right side.*
* *In this way, the largest element is moved to the rightmost end at first.*
* *This process is then continued to find the second largest and place it and so on until the data is sorted.*
* **Insertion Sort: Insertion sort**is a simple sorting algorithm that works by building a sorted array one element at a time. It is considered an ” **in-place**” sorting algorithm, meaning it doesn’t require any additional memory space beyond the original array.
* **Algorithm:**
* *Start with second element of the array as first element in the array is assumed to be sorted.*
* *Compare second element with the first element and check if the second element is smaller, then swap them.*
* *Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.*
* *Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.*
* *Repeat until the entire array is sorted.*
* **Quick Sort: Quicksort**is a sorting algorithm based on the Divide and Conquer Algorithm, that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. The key process in **quicksort**is a **partition()**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot. Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.
* **Algorithm:**

*QUICKSORT (array A, start, end)*

*{*

*1****if****(start < end)*

*2 {*

*3 p = partition(A, start, end)*

*4 QUICKSORT (A, start, p - 1)*

*5 QUICKSORT (A, p + 1, end)*

*6 }*

*}*

* **Merge Sort:** Merge sort is a sorting algorithm that follows the divide and conquerapproach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.
* **Algorithm:**
* **Divide:**Divide the list or array recursively into two halves until it can no more be divided.
* **Conquer:**Each subarray is sorted individually using the merge sort algorithm.
* **Merge:**The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

2. Setup:

**Class Creation**: create class **Product** with attributes for searching, such as **productId, productName**, and **category**.

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 +

                '}';

    }

}

* **Classes** :
* **Order Class:**
* **Attributes:**
  + orderId: An integer representing the unique ID of the order.
  + customerName: A string representing the name of the customer.
  + totalPrice: A double representing the total price of the order.
* **Constructor:**
  + Initializes the attributes with provided values.
* **toString Method:**
  + Overrides the toString method to return a string representation of the order details.
* **SortingCustomerOrders Class:**
* **Attributes:**
  + List<Order> orders: A list to store multiple Order objects.
* **Methods:**
  + **bubbleSort(List<Order> orders):**
    - Sorts the orders using the Bubble Sort algorithm based on the totalPrice.
  + **quickSort(List<Order> orders, int low, int high):**
    - Sorts the orders using the Quick Sort algorithm based on the totalPrice.
  + **partition(List<Order> orders, int low, int high):**
    - A helper method for the Quick Sort algorithm that partitions the list of orders around a pivot.
* **main Method:**
  + Prompts the user to enter the number of orders and the details for each order.
  + Prompts the user to choose a sorting method (Bubble Sort or Quick Sort).
  + Sorts the orders based on the chosen sorting method and prints the sorted orders.

3. Implementation:

* Implement Bubble Sort to sort orders by totalPrice:

 public static void bubbleSort(List<Order> orders) {

        int n = orders.size();

        for (int i = 0; i < n - 1; i++) {

            for (int j = 0; j < n - i - 1; j++) {

                if (orders.get(j).totalPrice > orders.get(j + 1).totalPrice) {

                    Order temp = orders.get(j);

                    orders.set(j, orders.get(j + 1));

                    orders.set(j + 1, temp);

                }

* Implement Quick Sort to sort orders by totalPrice:

public static void quickSort(List<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);

        }

    }

Here is the github repo link –

4. Analysis:

* Compare the performance (time complexity) of Bubble Sort and Quick Sort

**Time Complexity** of Bubble Sort**:**

* **Outer Loop:** The outer loop runs from i = 0 to i < n - 1, which results in n - 1 iterations.
* **Inner Loop:** For each iteration of the outer loop, the inner loop runs from j = 0 to j < n - i - 1. The number of iterations of the inner loop decreases with each pass of the outer loop.

**Total Comparisons:**

* In the first pass, the inner loop runs n - 1 times.
* In the second pass, it runs n - 2 times.
* This continues until the final pass, which runs 1 time.

The total number of comparisons is the sum of the series: (n−1)+(n−2)+… 1

This sum can be computed using the formula for the sum of the first n integers: Sum=((n−1)×n2)/2

* **Time Complexity:**
* **Worst-case Complexity:** O(n^2) – This occurs when the list is in reverse order, and every possible comparison and swap needs to be made.
* **Best-case Complexity:** O(n) – This occurs if the list is already sorted, as Bubble Sort can be optimized to stop early if no swaps are made during a pass. Without such optimization, the best-case complexity is still O(n2)O(n^2)O(n2).
* **Average-case Complexity:** O(n^2)– This is the expected complexity for most unsorted lists.

The Bubble Sort algorithm has a time complexity of O(n^2) in both average and worst cases due to its nested loop structure

**Time Complexity** of Quick Sort**:**

 The partition method runs in O(n) time.

 The recursion depth is typically log n in the average and best cases, leading to O(n log n) overall complexity.

 In the worst case, recursion depth can be n, leading to O(n^2)

* **Worst-case Complexity:** O(n^2) – This occurs when the pivot selection is poor and results in highly unbalanced partitions, such as when the smallest or largest element is chosen as the pivot
* **Best-case Complexity:** O(n log n) – On average, Quick Sort performs well if the pivot is chosen such that partitions are relatively balanced.
* **Average-case Complexity:** O(n log n)– This occurs when the pivot selection always results in perfectly balanced partitions.
* Discuss why Quick Sort is generally preferred over Bubble Sort.
* Efficiency

Bubble Sort:

Bubble Sort is straightforward but inefficient due to the large number of comparisons and swaps it performs. For every element, it repeatedly compares and swaps adjacent pairs, resulting in high overhead.

Quick Sort:

Quick Sort is more efficient because it reduces the problem size more rapidly with each partition. Instead of making multiple passes through the data, it divides the data into smaller sub-arrays and sorts them independently. This significantly reduces the number of comparisons and operations needed.

* Scalability

Bubble Sort:

Due to its O(n 2) time complexity, Bubble Sort does not scale well with larger datasets. Its performance degrades quickly as the size of the data increases.

Quick Sort:

Quick Sort is scalable and works well with larger datasets because of its O(n log n) average time complexity. It can handle large arrays efficiently and is commonly used in practice.

* Space Complexity

Bubble Sort:

Bubble Sort has a space complexity of O(1) because it is an in-place sorting algorithm that only requires a constant amount of additional memory for the swap operations.

Quick Sort:

Quick Sort also has an in-place sorting nature, with a space complexity of

O(log n) due to the recursive call stack. This is relatively efficient, though it can use more stack space than Bubble Sort in some cases.

* Practical Performance

Bubble Sort:

Despite its simplicity, Bubble Sort is rarely used in practice for large datasets because its performance is significantly poorer compared to more efficient algorithms.

Quick Sort:

Quick Sort is widely used in practice due to its efficient performance for large datasets. It is often the default choice in standard libraries for sorting (e.g., Java’s Arrays.sort() for primitive types uses Quick Sort).

* Summary:

Quick Sort is generally preferred over Bubble Sort because it offers better average and best-case time complexities, scales well with large datasets, and performs fewer operations compared to Bubble Sort. Although Quick Sort has a worst-case time complexity of 𝑂(𝑛2), this can be mitigated with optimizations. Overall, Quick Sort is a more efficient and scalable algorithm for sorting tasks.

**Output:**



