**Exercise 3: Sorting Customer Orders**

1. **Understand Sorting Algorithms:**

* *Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).*

Ans. Arranging the elements in ascending order or in descending order is called Sorting. Sorting techniques are broadly categorized into two.

Internal Sorting and External Sorting.

1. Internal Sorting : All the records that are to be sorted are in main memory.

2. External Sorting: Some sorts that cannot be performed in main memory and must be done on disk or tape. This type of sorting is known as External Sorting.

**Insertion Sort Algorithm:**

**Insertion Sort:**Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

**Algorithm**  
To sort an array of size n in ascending order:  
1: Iterate from arr[1] to arr[n] over the array.  
2: Compare the current element (key) to its predecessor.  
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

**Time Complexity of InsertionSort**

**Best Case :** **O(n)** #Means array is already sorted.

**Average Case :** **O(n²)** #Means array with random numbers.

**Worst Case : O(n²)** #Means array with descending order.

**Quick Sort Algorithm:**

**Quick Sort:**This is the best sort Technique. QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot
3. Pick a random element as pivot.
4. Pick median as pivot.

**Time Complexity :**

Best Case : **O(nlogn)** #Means array is already sorted.

Average Case : **O(nlogn)** #Means array with random numbers.

Worst Case :**O(n^2)** #Means array with descending order.

**Algorithm:**

1. First element is considered as pivot in the implemented code below.
2. Then we should move all the elements which are less than pivot to one side and greater than pivot to other side.
3. We divide the array into two arrays in such a way that elements > pivot and elements < pivot.

**Merge Sort Algorithm:**

**Merge Sort:**One of the best sorting technique. If n value is large, it follows divide and conquer approach.

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/" \t "_blank), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves.

**Time Complexity :**

Best Case : **O(nlogn)** #Means array is already sorted.

Average Case : **O(nlogn)** #Means array with random numbers.

Worst Case :**O(nlogn)** #Means array with descending order.

**Algorithm**:

**1.** Find the middle point to divide the array into two halves:   
 middle m = (l+r)/2  
**2.** Call mergeSort for first half:   
 Call mergeSort(arr, l, m)  
**3.** Call mergeSort for second half:  
 Call mergeSort(arr, m+1, r)  
**4.** Merge the two halves sorted in step 2 and 3:  
 Call merge(arr, l, m, r)

**Bubble Sort:**

A well known algorithm called Bubble sort, is easy to understand. Probably this is the least efficient. The basic idea underlying the bubble sort is to pass through the array left to right several times. Each pass consists of comparing each element in the array with its successor and interchanging the two elements if they are not in proper order. At the end of each pass we can observe that the largest element of the list is moved to its final position.

**Time Complexity :**

Best Case : **O(n²)** #Means array is already sorted.

Average Case : **O(n²)** #Means array with random numbers.

Worst Case :**O(n²)** #Means array with descending order.

1. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
2. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.

Code :

package Algorithm\_DataStructures.Exercise3.Sorting\_Customer\_Orders;

class Order {

    private int orderId;

    private String customerName;

    private double totalPrice;

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

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    public int getOrderId() {

        return orderId;

    }

    public String getCustomerName() {

        return customerName;

    }

    public double getTotalPrice() {

        return totalPrice;

    }

}

public class OrderSorting {

    // Bubble Sort to sort orders by totalPrice

    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

                    Order temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

    }

    // Quick Sort to sort orders by totalPrice

    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].getTotalPrice();

        int i = low - 1;

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

            if (orders[j].getTotalPrice() < 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(101, "Alice", 150.50),

            new Order(202, "Bob", 120.75),

            new Order(303, "Charlie", 200.25),

            new Order(404, "David", 180.30),

            new Order(505, "Eve", 100.80)

        };

        System.out.println("Orders before sorting:");

        for (Order order : orders) {

            System.out.println("Order ID: " + order.getOrderId() + ", Customer: " + order.getCustomerName() + ", Total Price: " + order.getTotalPrice());

        }

        // Sorting using Bubble Sort

        bubbleSort(orders);

        System.out.println("\nOrders after Bubble Sort:");

        for (Order order : orders) {

            System.out.println("Order ID: " + order.getOrderId() + ", Customer: " + order.getCustomerName() + ", Total Price: " + order.getTotalPrice());

        }

        Order[] ordersQuickSort = {

            new Order(101, "Alice", 150.50),

            new Order(202, "Bob", 120.75),

            new Order(303, "Charlie", 200.25),

            new Order(404, "David", 180.30),

            new Order(505, "Eve", 100.80)

        };

    // Sorting using Quick Sort

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

        System.out.println("\nOrders after Quick Sort:");

        for (Order order : ordersQuickSort) {

            System.out.println("Order ID: " + order.getOrderId() + ", Customer: " + order.getCustomerName() + ", Total Price: " + order.getTotalPrice());

        }

    }

}

1. **Analysis:**

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

Ans. **Time Complexity for Bubble Sort :**

Best Case : **O(n²)** #Means array is already sorted.

Average Case : **O(n²)** #Means array with random numbers.

Worst Case :**O(n²)** #Means array with descending order.

**Time Complexity for Quick Sort :**

Best Case : **O(nlogn)** #Means array is already sorted.

Average Case : **O(nlogn)** #Means array with random numbers.

Worst Case :**O(n^2)** #Means array with descending order.

* Discuss why Quick Sort is generally preferred over Bubble Sort.

Ans. Quick Sort is generally preferred over Bubble Sort for the following reasons:

* Efficiency: Quick Sort is more efficient, especially for large datasets.
* Simplicity: Bubble Sort is simpler, but Quick Sort is more powerful.
* Nearly Sorted Data: Quick Sort can be less optimal on nearly sorted data, but it is still generally preferred due to its overall performance.