NARESH M\_AIDS\_DAY-5\_PRACTICE

**1.Transition Point**

Given a sorted array, arr[] containing only 0s and 1s, find the transition point, i.e., the first index where 1 was observed, and before that, only 0 was observed.  If arr does not have any 1, return -1. If array does not have any 0, return 0.

Examples:

Input: arr[] = [0, 0, 0, 1, 1]

Output: 3

Explanation: index 3 is the transition point where 1 begins.

Input: arr[] = [0, 0, 0, 0]

Output: -1

Explanation: Since, there is no "1", the answer is -1.

Input: arr[] = [1, 1, 1]

Output: 0

Explanation: There are no 0s in the array, so the transition point is 0, indicating that the first index (which contains 1) is also the first position of the array.

**Program:**

import java.util.Scanner;

class transitionPoint {

    int transitionPoint(int arr[]) {

        int point = -1;

        for (int i = 0; i < arr.length; i++) {

            if (arr[i] == 1) {

                point = i;

                break;

            }

        }

        return point;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the array: ");

        int n = scanner.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements of the array : ");

        for (int i = 0; i < n; i++) {

            arr[i] = scanner.nextInt();

        }

        transitionPoint solution = new transitionPoint();

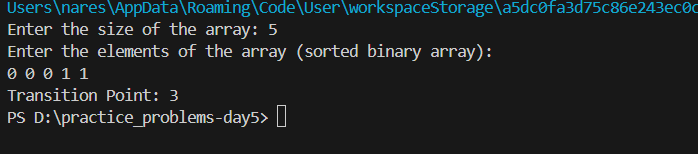
        int result = solution.transitionPoint(arr);

        System.out.println("Transition Point: " + result);

        scanner.close();

    }

}



**Time Complexity: O(n)**

**2.First Repeating Element**

Given an array **arr[],** find the first repeating element. The element should occur more than once and the index of its first occurrence should be the smallest.

**Note:-**The position you return should be according to 1-based indexing.

**Examples:**

**Input:** arr[] = [1, 5, 3, 4, 3, 5, 6]

**Output:** 2

**Explanation:** 5 appears twice and its first appearance is at index 2 which is less than 3 whose first the occurring index is 3.

**Input:** arr[] = [1, 2, 3, 4]

**Output:** -1

**Explanation:** All elements appear only once so answer is -1.

**Program:**

import java.util.HashMap;

import java.util.Scanner;

class firstRepeatingElement {

    // Function to return the position of the first repeating element.

    public static int firstRepeated(int[] arr) {

        HashMap<Integer, Integer> map = new HashMap<>();

        int min\_index = Integer.MAX\_VALUE;

        for (int i = 0; i < arr.length; i++) {

            if (map.containsKey(arr[i])) {

                min\_index = Math.min(min\_index, map.get(arr[i]));

            } else {

                map.put(arr[i], i + 1); // Storing 1-based index

            }

        }

        return min\_index == Integer.MAX\_VALUE ? -1 : min\_index;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the array: ");

        int n = scanner.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements of the array: ");

        for (int i = 0; i < n; i++) {

            arr[i] = scanner.nextInt();

        }

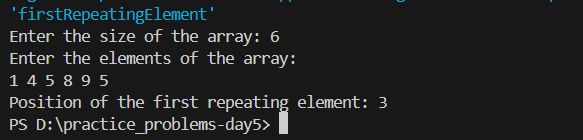
        int result = firstRepeated(arr);

        System.out.println("Position of the first repeating element: " + result);

        scanner.close();

    }

}

****

**Time Complexity: O(n)**

**3.Stock Buy and sell**

The cost of stock on each day is given in an array **A**[] of size **N**. Find all the segments of days on which you buy and sell the stock such that the sum of difference between sell and buy prices is maximized. Each segment consists of indexes of two elements, first is index of day on which you buy stock and second is index of day on which you sell stock.  
**Note:** Since there can be multiple solutions, the driver code will print 1 if your answer is correct, otherwise, it will return 0. In case there's no profit the driver code will print the string "**No Profit**" for a correct solution.  
  
**Example 1:**

**Input:**

N = 7

A[] = {100,180,260,310,40,535,695}

**Output:**

1

**Explanation:**

One possible solution is (0 3) (4 6)

We can buy stock on day 0,

and sell it on 3rd day, which will

give us maximum profit. Now, we buy

stock on day 4 and sell it on day 6.

**Program:**

import java.util.ArrayList;

import java.util.Scanner;

class stockBuyAndSell {

    ArrayList<ArrayList<Integer>> stockBuySell(int A[], int n) {

        ArrayList<ArrayList<Integer>> result = new ArrayList<>();

        int i = 0;

        while (i < n - 1) {

            while (i < n - 1 && A[i + 1] <= A[i]) {

                i++;

            }

            if (i == n - 1) {

                break;

            }

            int buy = i++;

            while (i < n && A[i] >= A[i - 1]) {

                i++;

            }

            int sell = i - 1;

            ArrayList<Integer> pair = new ArrayList<>();

            pair.add(buy);

            pair.add(sell);

            result.add(pair);

        }

        return result;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of days (N): ");

        int n = sc.nextInt();

        int[] A = new int[n];

        System.out.print("Enter the stock prices: ");

        for (int i = 0; i < n; i++) {

            A[i] = sc.nextInt();

        }

        stockBuyAndSell solution = new stockBuyAndSell();

        ArrayList<ArrayList<Integer>> result = solution.stockBuySell(A, n);

        if (result.isEmpty()) {

            System.out.println("No Profit");

        } else {

            System.out.println("The buy-sell pairs are:");

            for (ArrayList<Integer> pair : result) {

                System.out.println(pair.get(0) + " " + pair.get(1));

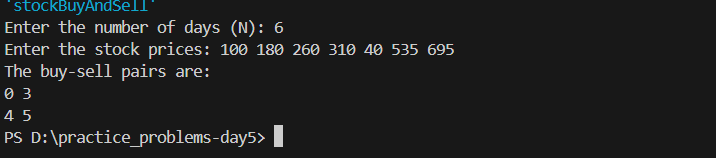
            }

        }

        sc.close();

    }

}



**Time complexity: O(n)**

**4. Remove duplicates in a sorted array:**Given a **sorted** array**arr.** Return the size of the modified array which contains only distinct elements.  
*Note:*  
1.Don't use set or HashMap to solve the problem.  
2. You **must** return the modified array **size only**where distinct elements are present and **modify** the original array such that all the distinct elements come at the beginning of the original array.

**Examples :**

**Input:** arr = [2, 2, 2, 2, 2]

**Output:** [2]

**Explanation:** After removing all the duplicates only one instance of 2 will remain i.e. [2] so modified array will contains 2 at first position and you should **return 1** after modifying the array, the driver code will print the modified array elements.

**Input:** arr = [1, 2, 4]

**Output:** [1, 2, 4]  
**Explation:** As the array does not contain any duplicates so you should return 3.

**Constraints:**  
1 ≤ arr.size() ≤ 105  
1 ≤ ai ≤ 106

**Program:**

import java.util.List;

import java.util.ArrayList;

import java.util.Scanner;

class removeDuplicatesInSortedArray {

    // Function to remove duplicates from the given list

    public int remove\_duplicate(List<Integer> arr) {

        if (arr.size() == 0) {

            return 0;

        }

        int j = 0;

        for (int i = 1; i < arr.size(); i++) {

            if (!arr.get(i).equals(arr.get(i - 1))) {

                j++;

                arr.set(j, arr.get(i));

            }

        }

        return j + 1;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the size of the array: ");

        int n = sc.nextInt();

        List<Integer> arr = new ArrayList<>();

        System.out.println("Enter the elements of the array: ");

        for (int i = 0; i < n; i++) {

            arr.add(sc.nextInt());

        }

        removeDuplicatesInSortedArray solution = new removeDuplicatesInSortedArray();

        int uniqueCount = solution.remove\_duplicate(arr);

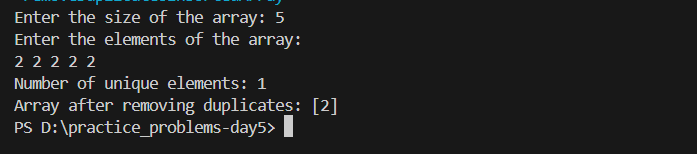
        System.out.println("Number of unique elements: " + uniqueCount);

        System.out.println("Array after removing duplicates: " + arr.subList(0, uniqueCount));

        sc.close();

    }

}



**Time complexity: O(n)**

**5.Wave Array**

Given a sorted array arr[] of distinct integers. Sort the array into a wave-like array(In Place). In other words, arrange the elements into a sequence such that arr[1] >= arr[2] <= arr[3] >= arr[4] <= arr[5].....  
If there are multiple solutions, find the lexicographically smallest one.

Note: The given array is sorted in ascending order, and you don't need to return anything to change the original array.

Examples:

Input: arr[] = [1, 2, 3, 4, 5]

Output: [2, 1, 4, 3, 5]

Explanation: Array elements after sorting it in the waveform are 2, 1, 4, 3, 5.

Input: arr[] = [2, 4, 7, 8, 9, 10]

Output: [4, 2, 8, 7, 10, 9]

Explanation: Array elements after sorting it in the waveform are 4, 2, 8, 7, 10, 9.  
  
Input: arr[] = [1]  
Output: [1]

Constraints:  
1 ≤ arr.size ≤ 106  
0 ≤ arr[i] ≤107

**program:**

import java.util.Scanner;

public class waveArray {

    public static void waveConvert(int[] arr) {

        int i = 0, j = 1;

        while (i < arr.length && j < arr.length) {

            int temp = arr[i];

            arr[i] = arr[j];

            arr[j] = temp;

            i += 2;

            j += 2;

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of elements: ");

        int n = sc.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter elements:");

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        waveConvert(arr);

        for (int i = 0; i < n; i++) {

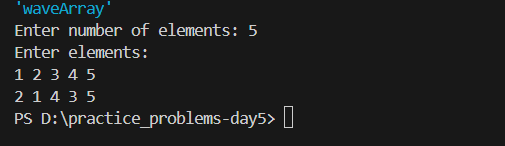
            System.out.print(arr[i] + " ");

        }

        sc.close();

    }

}



Time Complexity: O(n)

**6.Coin change**

Given an integer array coins[ ] representing different denominations of currency and an integer sum, find the number of ways you can make sum by using different combinations from coins[ ].   
Note: Assume that you have an infinite supply of each type of coin. And you can use any coin as many times as you want.  
Answers are guaranteed to fit into a 32-bit integer.

Examples:

Input: coins[] = [1, 2, 3], sum = 4

Output: 4

Explanation: Four Possible ways are: [1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3].

**Program:**

import java.util.\*;

class coinChange{

    public static int minCoins(int[] coins, int sum) {

        int[] dp = new int[sum + 1];

        Arrays.fill(dp, Integer.MAX\_VALUE);

        dp[0] = 0;

        for (int coin : coins) {

            for (int i = coin; i <= sum; i++) {

                if (dp[i - coin] != Integer.MAX\_VALUE) {

                    dp[i] = Math.min(dp[i], dp[i - coin] + 1);

                }

            }

        }

        return dp[sum] == Integer.MAX\_VALUE ? -1 : dp[sum];

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the number of coins:");

        int n = scanner.nextInt();

        int[] coins = new int[n];

        System.out.println("Enter the coin denominations:");

        for (int i = 0; i < n; i++) {

            coins[i] = scanner.nextInt();

        }

        System.out.println("Enter the target sum:");

        int sum = scanner.nextInt();

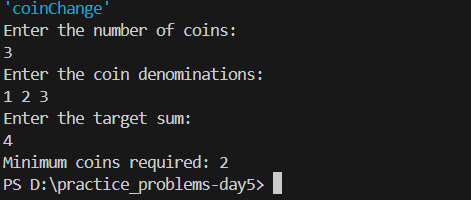
        int result = minCoins(coins, sum);

        System.out.println("Minimum coins required: " + result);

        scanner.close();

    }

}



**Time complexity: O(m x sum)**

**7. First and Last Occurrences**

Given a sorted array arr with possibly some duplicates, the task is to find the first and last occurrences of an element x in the given array.  
Note: If the number x is not found in the array then return both the indices as -1.

Examples:

Input: arr[] = [1, 3, 5, 5, 5, 5, 67, 123, 125], x = 5

Output: [2, 5]

Explanation: First occurrence of 5 is at index 2 and last occurrence of 5 is at index 5

**Program:**

import java.util.Scanner;

public class firstAndLastOccurence {

    public static int[] findFirstAndLast(int arr[], int x) {

        int[] result = {-1, -1};

        result[0] = findFirstOccurrence(arr, x);

        result[1] = findLastOccurrence(arr, x);

        return result;

    }

    public static int findFirstOccurrence(int arr[], int x) {

        int l = 0, r = arr.length - 1, first = -1;

        while (l <= r) {

            int mid = l + (r - l) / 2;

            if (arr[mid] == x) {

                first = mid;

                r = mid - 1;

            } else if (arr[mid] < x) {

                l = mid + 1;

            } else {

                r = mid - 1;

            }

        }

        return first;

    }

    public static int findLastOccurrence(int arr[], int x) {

        int l = 0, r = arr.length - 1, last = -1;

        while (l <= r) {

            int mid = l + (r - l) / 2;

            if (arr[mid] == x) {

                last = mid;

                l = mid + 1;

            } else if (arr[mid] < x) {

                l = mid + 1;

            } else {

                r = mid - 1;

            }

        }

        return last;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of elements in the array: ");

        int n = sc.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements in the sorted array: ");

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        System.out.print("Enter the element to find: ");

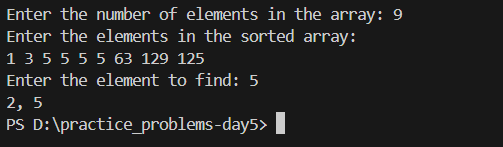
        int x = sc.nextInt();

        int[] result = findFirstAndLast(arr, x);

        System.out.println(result[0] + ", " + result[1]);

    }

}



**Time Complexity: O (log n)**

**8.Maximum Index**

Given an array arr of positive integers. The task is to return the maximum of j - i subjected to the constraint of arr[i] < arr[j] and i < j.

Examples:

Input: arr[] = [1, 10]

Output: 1

Explanation: arr[0] < arr[1] so (j-i) is 1-0 = 1.

**Program:**

import java.util.Scanner;

class maximumIndex {

    int maxIndexDiff(int[] arr) {

        int n = arr.length;

        if (n == 1) {

            return 0;

        }

        int diff= -1;

        int[] leftMin = new int[n];

        int[] rightMax = new int[n];

        leftMin[0] = arr[0];

        for (int l = 1; l < n; ++l) {

            leftMin[l] = Math.min(arr[l], leftMin[l - 1]);

        }

        rightMax[n - 1] = arr[n - 1];

        for (int r = n - 2; r >= 0; --r) {

            rightMax[r] = Math.max(arr[r], rightMax[r + 1]);

        }

        int i = 0, j = 0;

        while (i < n && j < n) {

            if (leftMin[i] <= rightMax[j]) {

                diff = Math.max(diff, j - i);

                j++;

            } else {

                i++;

            }

        }

        return diff;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of elements in the array: ");

        int n = sc.nextInt();

        int[] arr = new int[n];

        System.out.print("Enter the elements of the array: ");

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

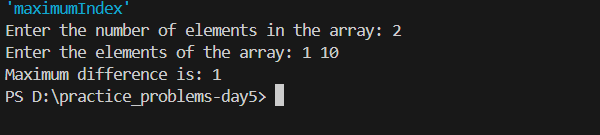
        maximumIndex solution = new maximumIndex();

        System.out.println("Maximum difference is: " + solution.maxIndexDiff(arr));

        sc.close();

    }

}



**Time complexity: O(n)**