

# DSA PRACTICE DAY4(14.11.24)

Name :Sundara Vinayagam V

Dept: B.Tech- CSBS

## 1.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.

### Example 2:

#### Input:

N = 5

A[] = {4,2,2,2,4}

#### Output:

1

#### Explanation:

There are multiple possible solutions.  
one of them is (3 4)

We can buy stock on day 3,  
and sell it on 4th day, which will  
give us maximum profit.

**Your Task:**

The task is to complete the function **stockBuySell()** which takes an array of A[] and N as input parameters and finds the days of buying and selling stock. The function must return a 2D list of integers containing all the buy-sell pairs i.e. the first value of the pair will represent the day on which you buy the stock and the second value represent the day on which you sell that stock. If there is No Profit, return an empty list.

**Expected Time Complexity:** O(N)

**Expected Auxiliary Space:** O(N)

**Constraints:**

$$2 \leq N \leq 10^6$$

$$0 \leq A[i] \leq 10^6$$

**Solution:**

```
class Solution{

    //Function to find the days of buying and selling stock for max profit.

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

        // code here

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

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

            if(A[i+1]>A[i]){

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

                al.add(i);

                al.add(i+1);

                a.add(al);

            }



        }

        return a;

    }

}
```

**Output:**

For Input:  

7

100 180 260 310 40 535 695

Your Output:

1

Expected Output:

1

**Time Complexity:**

**Space Complexity:**

## 2.Coin Change (Count Ways)

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].

**Input:** coins[] = [2, 5, 3, 6], sum = 10

**Output:** 5

**Explanation:** Five Possible ways are: [2, 2, 2, 2, 2], [2, 2, 3, 3], [2, 2, 6], [2, 3, 5] and [5, 5].

**Input:** coins[] = [5, 10], sum = 3

**Output:** 0

**Explanation:** Since all coin denominations are greater than sum, no combination can make the target sum.

**Constraints:**

$1 \leq \text{sum} \leq 1e4$

$1 \leq \text{coins}[i] \leq 1e4$

$1 \leq \text{coins.size()} \leq 1e3$

**Solution:****Output:****Time Complexity:****Space Complexity:****3.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

**Input:** `arr[] = [1, 3, 5, 5, 5, 5, 7, 123, 125]`, `x = 7`

**Output:** `[6, 6]`

**Explanation:** First and last occurrence of 7 is at index 6

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

**Output:** `[-1, -1]`

**Explanation:** No occurrence of 4 in the array, so, output is `[-1, -1]`

**Constraints:**

$1 \leq \text{arr.size()} \leq 10^6$

$1 \leq \text{arr}[i], x \leq 10^9$

**Solution:**

```
class GFG {  
    ArrayList<Integer> find(int arr[], int x) {  
        int f=-1,l=-1;
```

```

    ArrayList<Integer> r=new ArrayList<Integer>();
    for(int i=0;i<arr.length;i++){
        if(arr[i]==x){
            f=i;
            break;
        }
    }
    for(int i=arr.length-1;i>=0;i--){
        if(arr[i]==x){
            l=i;
            break;
        }
    }
    r.add(f);
    r.add(l);
    return r;
}
}

```

### Output:

```

For Input: [1, 3, 5, 5, 5, 5, 6, 7, 12, 3, 12, 5]
5

Your Output:
2 5

Expected Output:
2 5

```

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

**Space Complexity:** $O(n)$

#### 4.Find 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.

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

**Output:** 1

**Explanation:** Index 1 is the transition point where 1 starts, and before it, only 0 was observed.

**Constraints:**

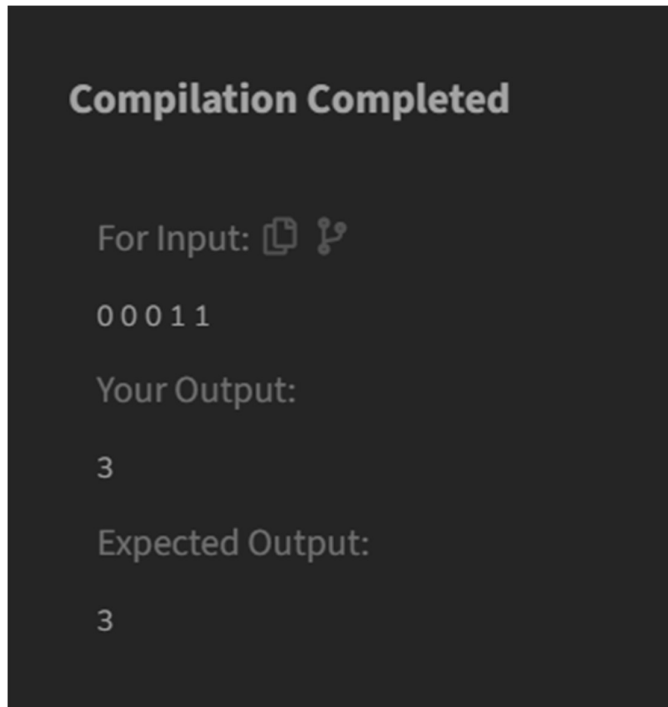
$1 \leq \text{arr.size()} \leq 10^5$

$0 \leq \text{arr}[i] \leq 1$

**Solution:**

```
class Solution {
    int transitionPoint(int arr[]) {
        for(int i=0;i<arr.length;i++){
            if(arr[i]==1){
                return i;
            }
        }
        return -1;
    }
}
```

**Output:**



**Time Complexity:** $O(N)$

**Space Complexity:** $O(n)$

### 5.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.

**Constraints:**



$1 \leq \text{arr.size} \leq 10^6$

$0 \leq \text{arr}[i] \leq 10^6$

**Solution:**

```
class Solution {  
    // Function to return the position of the first repeating element.  
    public static int firstRepeated(int[] arr) {  
        HashMap<Integer, Integer> map = new HashMap<>();  
        int index = -1;  
        for(int x : arr)  
            map.put(x, map.getOrDefault(x,0)+1);  
        for(int i = 0; i<arr.length; i++){  
            if(map.get(arr[i])>1){  
                index = i+1;  
                break;  
            }  
        }  
        return index;  
    }  
}
```

**Output:**

For Input:  

1 5 3 4 3 5 6

Your Output:

2

Expected Output:

2



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

**Space Complexity:** $O(n)$

### 6.Remove Duplicates 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]

**Explanation:** As the array does not contain any duplicates so you should return 3.

**Constraints:**

$1 \leq \text{arr.size()} \leq 10^5$

$1 \leq a_i \leq 10^6$

**Solution:**

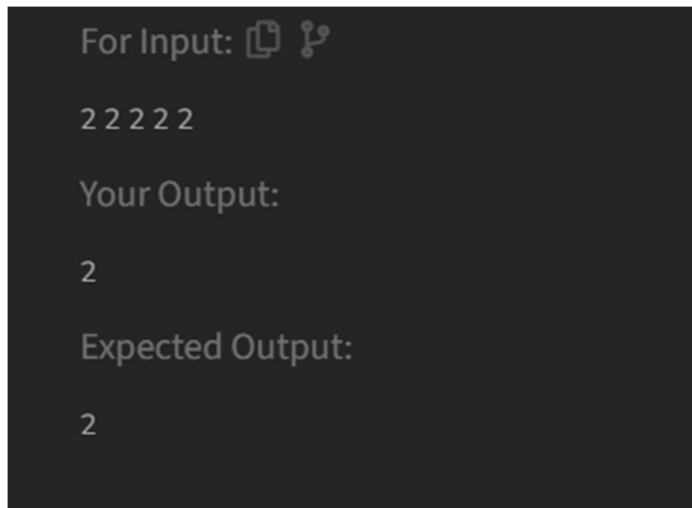
```
class Solution {  
    public int remove_duplicate(List<Integer> arr) {  
        if(arr.size() ==0){  
            return 0;  
        }  
        int u=0;  
        for (int i = 1; i < arr.size(); i++) {  
            if (!arr.get(i).equals(arr.get(u))) {  
                u++;  
                arr.set(u, arr.get(i));  
            }  
        }  
    }  
}
```

```
    return u + 1;

}

}
```

**Output:**



**Time Complexity:** $O(N)$

**Space Complexity:** $O(N)$

### 7. 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.

**Input:** arr[] = [34, 8, 10, 3, 2, 80, 30, 33, 1]

**Output:** 6

**Explanation:** In the given array arr[1] < arr[7] satisfying the required condition(arr[i] ≤ arr[j]) thus giving the maximum difference of j - i which is 6(7-1).

**Expected Time Complexity:**  $O(n)$

**Expected Auxiliary Space:**  $O(n)$

**Constraints:** $1 \leq \text{arr.size} \leq 10^6$  $0 \leq \text{arr}[i] \leq 10^9$ **Solution:**

```
import java.util.*;

public class MaxIndex {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int t = Integer.parseInt(scanner.nextLine().trim());

        while (t-- > 0) {
            String line = scanner.nextLine().trim();
            String[] numsStr = line.split(" ");
            int[] nums = new int[numsStr.length];
            for (int i = 0; i < numsStr.length; i++) {
                nums[i] = Integer.parseInt(numsStr[i]);
            }



            int n = nums.length;
            int[] minLeft = new int[n];
            int[] maxRight = new int[n];
            minLeft[0] = nums[0];
            for (int i = 1; i < n; i++) {
                minLeft[i] = Math.min(nums[i], minLeft[i - 1]);
            }
            maxRight[n - 1] = nums[n - 1];
            for (int j = n - 2; j >= 0; j--) {
                maxRight[j] = Math.max(nums[j], maxRight[j + 1]);
            }

            int i = 0;
            int j = 0;
            int maxDiff = -1;
            while (i < n && j < n) {
                if (minLeft[i] <= maxRight[j]) {
                    maxDiff = Math.max(maxDiff, j - i);
                    j++;
                } else {
                    i++;
                }
            }

            System.out.println(maxDiff);
        }
    }
}
```

```
}
```

**Output:**

```
For Input:    
1 10  
Your Output:  
1  
Expected Output:  
1
```

**Time Complexity:**

**Space Complexity:**

### 8.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  $arr[1] \geq arr[2] \leq arr[3] \geq arr[4] \leq arr[5] \dots$

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:**


$1 \leq arr.size \leq 10^6$

$0 \leq arr[i] \leq 10^7$

**Solution:**

```
class Solution {  
    public static void convertToWave(int[] arr) {  
        int i=0;  
        while(i<arr.length-1){  
            int t=arr[i];  
            arr[i]=arr[i+1];  
            arr[i+1]=t;  
            i+=2;  
        }  
    }  
}
```

**Output:**

For Input:  

1 2 3 4 5

Your Output:

2 1 4 3 5

Expected Output:

2 1 4 3 5

**Time Complexity:** $O(N)$

**Space Complexity:** $O(N)$