

CODING PRACTICE-5(14.11.2024)

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.

Solution:

```
import java.io.*;
import java.util.*;
public class Buy1
{
    public static void main (String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int t = Integer.parseInt(br.readLine().trim());
        while(t-->0){
            int n = Integer.parseInt(br.readLine().trim());
            int A[] = new int[n];
            String inputLine[] = br.readLine().trim().split(" ");
            for(int i=0; i<n; i++){
                A[i] = Integer.parseInt(inputLine[i]);
            }
            int p = 0;
            for(int i=0; i<n-1; i++)
                p += Math.max(0,A[i+1]-A[i]);
            Solution obj = new Solution();
            ArrayList<ArrayList<Integer> > ans = obj.stockBuySell(A, n);
            if(ans.size()==0)
                System.out.print("No Profit");
            else{
                int c=0;
                for(int i=0; i<ans.size(); i++)
                    c += A[ans.get(i).get(1)]-A[ans.get(i).get(0)];
            }
        }
    }
}
```

```

        if(c==p)
            System.out.print(1);
        else
            System.out.print(0);
    }System.out.println();

System.out.println("~");
}
}
}
class Solution{
    ArrayList<ArrayList<Integer>> stockBuySell(int A[], int n) {
        ArrayList<ArrayList<Integer>> ans= new ArrayList<>();
        for(int i=0;i<A.length-1;i++){
            if(A[i]<A[i+1]){
                ArrayList<Integer> ar=new ArrayList<>();
                ar.add(i);
                ar.add(i+1);
                ans.add(ar);
            }
        }
        return ans;
    }
}

```

Output:

```
C:\Windows\System32\cmd.e  X  +  v
Microsoft Windows [Version 10.0.22631.4460]
(c) Microsoft Corporation. All rights reserved.

F:\>javac Buy.java

F:\>java Buy
1
6
45 34 55 65 20 24
1
~
```

Time complexity: $O(n)$

Space complexity: $O(n)$

2. 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]`

Solution:

```
import java.io.*;
```

```
import java.util.*;
```

```

import java.util.Arrays;

class GFG {

    public static int[] input(BufferedReader br, int n) throws IOException {

        String[] s = br.readLine().trim().split(" ");

        int[] a = new int[n];

        for (int i = 0; i < n; i++) a[i] = Integer.parseInt(s[i]);

        return a;

    }

    public static void print(int[] a) {

        for (int e : a) System.out.print(e + " ");

        System.out.println();

    }

    public static void print(ArrayList<Integer> a) {

        for (int e : a) System.out.print(e + " ");

        System.out.println();

    }

}

class Wave {

    public static void main(String[] args) throws IOException {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int t;

        t = Integer.parseInt(br.readLine());

        while (t-- > 0) {

            String inputLine[] = br.readLine().trim().split(" ");

            int n = inputLine.length;

            int arr[] = new int[n];

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

                arr[i] = Integer.parseInt(inputLine[i]);

            }

        }

    }

}

```

```

        Solution obj = new Solution();
        obj.convertToWave(arr);
        StringBuffer sb = new StringBuffer("");
        for (int i : arr) {
            sb.append(i + " ");
        }
        System.out.println(sb.toString());
        System.out.println("~");
    }
}
class Solution {
    public static void convertToWave(int[] arr) {
        for(int i=0;i<arr.length-1;i+=2){
            int temp=arr[i];
            arr[i]=arr[i+1];
            arr[i+1]=temp;
        }
    }
}

```

Output:

```

F:\>javac Wave.java

F:\>java Wave
1
2 4 3 1 5 3
4 2 1 3 3 5
~

F:\>

```

Time complexity : $O(n)$

Space Complexity: $O(1)$

3. 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 contain 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.

Solution:

```
import java.io.*;
import java.util.*;

public class Duplicate{

    public static void main(String[] args) throws Exception {

        Scanner sc = new Scanner(System.in);

        int t = Integer.parseInt(sc.nextLine());

        while (t-- > 0) {

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

            String input = sc.nextLine();

            StringTokenizer st = new StringTokenizer(input);
```

```

        while (st.hasMoreTokens()) {
            arr.add(Integer.parseInt(st.nextToken()));
        }
        Solution ob = new Solution();
        int ans = ob.remove_duplicate(arr);
        for (int i = 0; i < ans; i++) {
            System.out.print(arr.get(i) + " ");
        }
        System.out.println();
        System.out.println("~");
    }
    sc.close();
}

class Solution {
    public int remove_duplicate(List<Integer> arr) {
        LinkedHashSet<Integer> n = new LinkedHashSet<>(arr);
        arr.clear();
        arr.addAll(n);
        return n.size();
    }
}

```

Output:


```
C:\Windows\System32\cmd.e  ×  +  ∨  
Microsoft Windows [Version 10.0.22631.4460]  
(c) Microsoft Corporation. All rights reserved.  
  
F:\>javac Duplicate.java  
  
F:\>java Duplicate  
1  
2 2 2 2 2  
2  
~  
  
F:\>
```

Time complexity: $O(n)$

Space Complexity: $O(n)$

4.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.

Solution:

```
import java.io.*;
```

```
import java.util.*;
```

```
class FirstRepeat{
```

```
    public static void main(String[] args) throws IOException {
```

```

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int t = Integer.parseInt(br.readLine().trim()); // Inputting the testcases

while (t-- > 0) {

    String line = br.readLine();

    String[] tokens = line.split(" ");

    ArrayList<Long> array = new ArrayList<>();

    for (String token : tokens) {

        array.add(Long.parseLong(token));

    }

    int[] arr = new int[array.size()];

    int idx = 0;

    for (long i : array) arr[idx++] = (int)i;

    Solution obj = new Solution();

    System.out.println(obj.firstRepeated(arr));

    System.out.println("~");

}

}

}

class Solution {

public static int firstRepeated(int[] arr) {

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

    int min = Integer.MAX_VALUE;

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

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

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

        }

        else{

            map.put(arr[i],i);

        }

    }

}

}

```

```

    }
}
return (min == Integer.MAX_VALUE) ? -1 : min+1;
}
}

```

Output:

```

C:\Windows\System32\cmd.e
F:\day-5>javac FirstRepeat.java
F:\day-5>java FirstRepeat
1
1 3 4 5 6 3
2
~
F:\day-5>

```

Time Complexity: $O(n)$

Space Complexity: $O(n)$

5. 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.

Solution:

```
import java.io.*;
import java.util.*;
class Transition {
    public static void main(String args[]) throws IOException {
        BufferedReader read = new BufferedReader(new InputStreamReader(System.in));
        int t = Integer.parseInt(read.readLine());
        while (t > 0) {
            String inputLine[] = read.readLine().trim().split(" ");
            int n = inputLine.length;
            int arr[] = new int[n];
            for (int i = 0; i < n; i++) {
                arr[i] = Integer.parseInt(inputLine[i]);
            }
            Solution obj = new Solution();
            System.out.println(obj.transitionPoint(arr));
            System.out.println("~");
            t--;
        }
    }
}

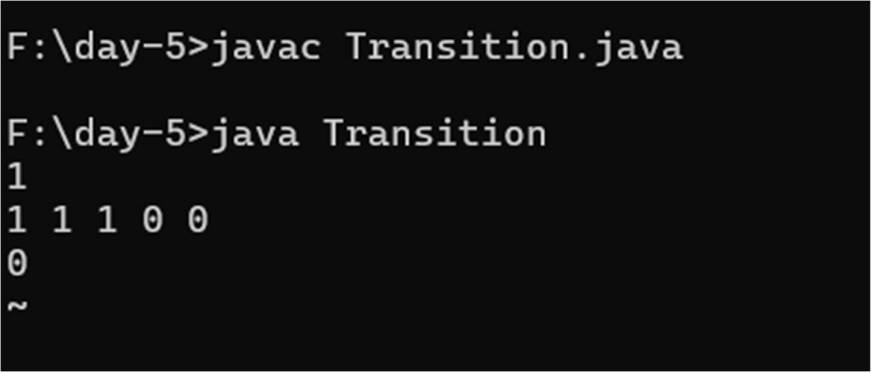
class Solution {
    int transitionPoint(int arr[]) {
        int left, right, mid, n, index;
```

```

n = arr.length;
left = 0;
right = n-1;
index = -1;
while(left <= right){
    mid = (left+right)/2;
    if(arr[mid] == 1){
        index = mid;
        right = mid-1;
    }
    else if(arr[mid] == 0){
        left = mid+1;
    }
}
return index;
}
}

```

Output:



```

F:\day-5>javac Transition.java

F:\day-5>java Transition
1
1 1 1 0 0
0
~

```

Time Complexity: $O(\log n)$

Space Complexity: $O(1)$

6. 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.

Solution:

```
import java.io.*;
```

```
import java.util.*;
```

```
class Coin{
```

```
    public static void main(String args[]) throws IOException {
```

```
        BufferedReader read = new BufferedReader(new InputStreamReader(System.in));
```

```
        int t = Integer.parseInt(read.readLine());
```

```
        while (t-- > 0) {
```

```
            String inputLine[] = read.readLine().trim().split(" ");
```

```
            int n = inputLine.length;
```

```
            int arr[] = new int[n];
```

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

```
                arr[i] = Integer.parseInt(inputLine[i]);
```

```
            }
```

```

        int sum = Integer.parseInt(read.readLine());

        Solution ob = new Solution();

        System.out.println(ob.count(arr, sum));
    }
}
}

class Solution {

    public static long solve(int ind, int target, int coins[], long dp[][]){
        if(target==0) return 1;
        if(target<0 || ind<0) return 0;
        if(dp[ind][target] != -1) return dp[ind][target];
        long nTake = solve(ind-1, target, coins, dp);
        long take = solve(ind, target-coins[ind], coins, dp);
        return dp[ind][target] = (take+nTake);
    }

    long count(int coins[], int sum) {
        int n = coins.length;
        long dp[][] = new long[n][sum+1];
        for(long row[]:dp)
            Arrays.fill(row, -1);
        return solve(n-1, sum, coins, dp);
    }
}

```

Output:

```

F:\day-5>javac Coin.java
F:\day-5>java Coin
1
1 2 3 4
5
6

```

TimeComplexity: $O(n*\text{sum})$

Space Complexity: $O(n*\text{sum})$

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

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]

Solution:

```
import java.io.*;
import java.util.*;
class GFG {
    public static int lastOccurenceOptimal(int arr[], int key) {
        int n=arr.length;
        int start = 0;
        int end = n-1;
        int result = -1;
        while (start <= end) {
            int mid = start + (end - start) / 2;
            if (arr[mid] == key) {
```



```

        result = mid;
        start = mid + 1;

    } else if (arr[mid] < key) {
        start = mid + 1;

    } else {
        end = mid - 1;

    }

}

return result;
}

public static int firstOccurenceOptimal(int arr[], int key) {
    int n = arr.length;
    int start = 0;
    int end = n - 1;
    int ans = -1;
    while (start <= end) {
        int mid = start + (end - start) / 2;
        if (arr[mid] == key) {
            ans = mid;
            end = mid - 1;
        } else if (arr[mid] < key) {
            start = mid + 1;
        } else {
            end = mid - 1;
        }
    }
}

```

```

    }

    return ans;
}

ArrayList<Integer> find(int arr[], int x) {
    ArrayList<Integer> result = new ArrayList<>();
    result.add(firstOccurenceOptimal(arr, x));
    result.add(lastOccurenceOptimal(arr, x));
    return result;
}
}

class Array {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int testcases = Integer.parseInt(br.readLine());
        while (testcases-- > 0) {
            String line1 = br.readLine();
            String[] a1 = line1.trim().split("\\s+");
            int n = a1.length;
            int arr[] = new int[n];
            for (int i = 0; i < n; i++) {
                arr[i] = Integer.parseInt(a1[i]);
            }
            int x = Integer.parseInt(br.readLine());
            GFG ob = new GFG();
            ArrayList<Integer> ans = ob.find(arr, x);
            System.out.println(ans.get(0) + " " + ans.get(1));
            System.out.println("~");
        }
    }
}

```

```
}  
}
```

Output:

```
F:\day-5>javac 0c.java
```

```
F:\day-5>java 0c
```

```
1
```

```
1 2 3
```

```
4
```

```
-1 -1
```

```
~
```

TimeComplexity: $O(\log n)$

Space Complexity: $O(1)$

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 $arr[i] \leq arr[j]$ and $i \leq j$.

Examples:

Input: arr[] = [1, 10]

Output: 1

Explanation: $arr[0] \leq arr[1]$ 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 ($arr[i] \leq arr[j]$) thus giving the maximum difference of $j - i$ which is $6(7-1)$.

Solution:

```
import java.io.*;
```

```
import java.util.*;
```

```
public class Maximum1 {
```

```
    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]);
    }
    Solution ob = new Solution();
    System.out.println(ob.maxIndexDiff(nums));
System.out.println("~");
}
}
}

class Solution {
    int maxIndexDiff(int[] arr) {
        // Your code here
        int n=arr.length;
        int minLeft[]=new int[n];
        int maxRight[]=new int[n];
        minLeft[0]=arr[0];
        for(int i=1;i<n;i++){
            minLeft[i]=Math.min(arr[i],minLeft[i-1]);
        }
        maxRight[n-1]=arr[n-1];
        for(int j=n-2;j>=0;j--){
            maxRight[j]=Math.max(arr[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++;
    }
}
return maxdiff;
}
}

```

Output:

```

F:\day-5>javac Maximum1.java

F:\day-5>java Maximum1
1
1 10
1
~

```

Time Complexity: $O(n)$

SpaceComplexity: $O(n)$