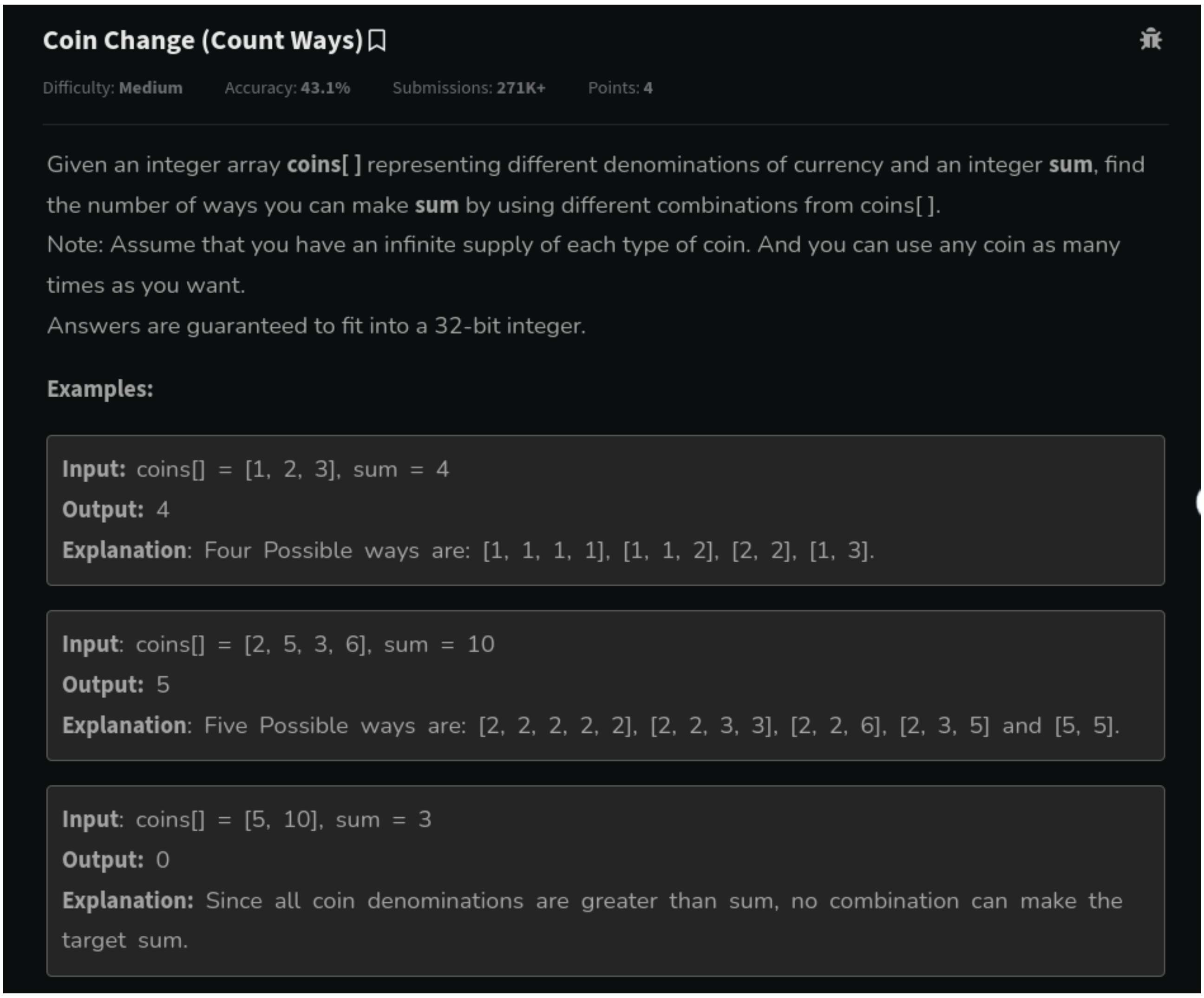


Coin Change (Count Ways)

Difficulty: **Medium** Accuracy: **43.1%** Submissions: **271K+** Points: **4**

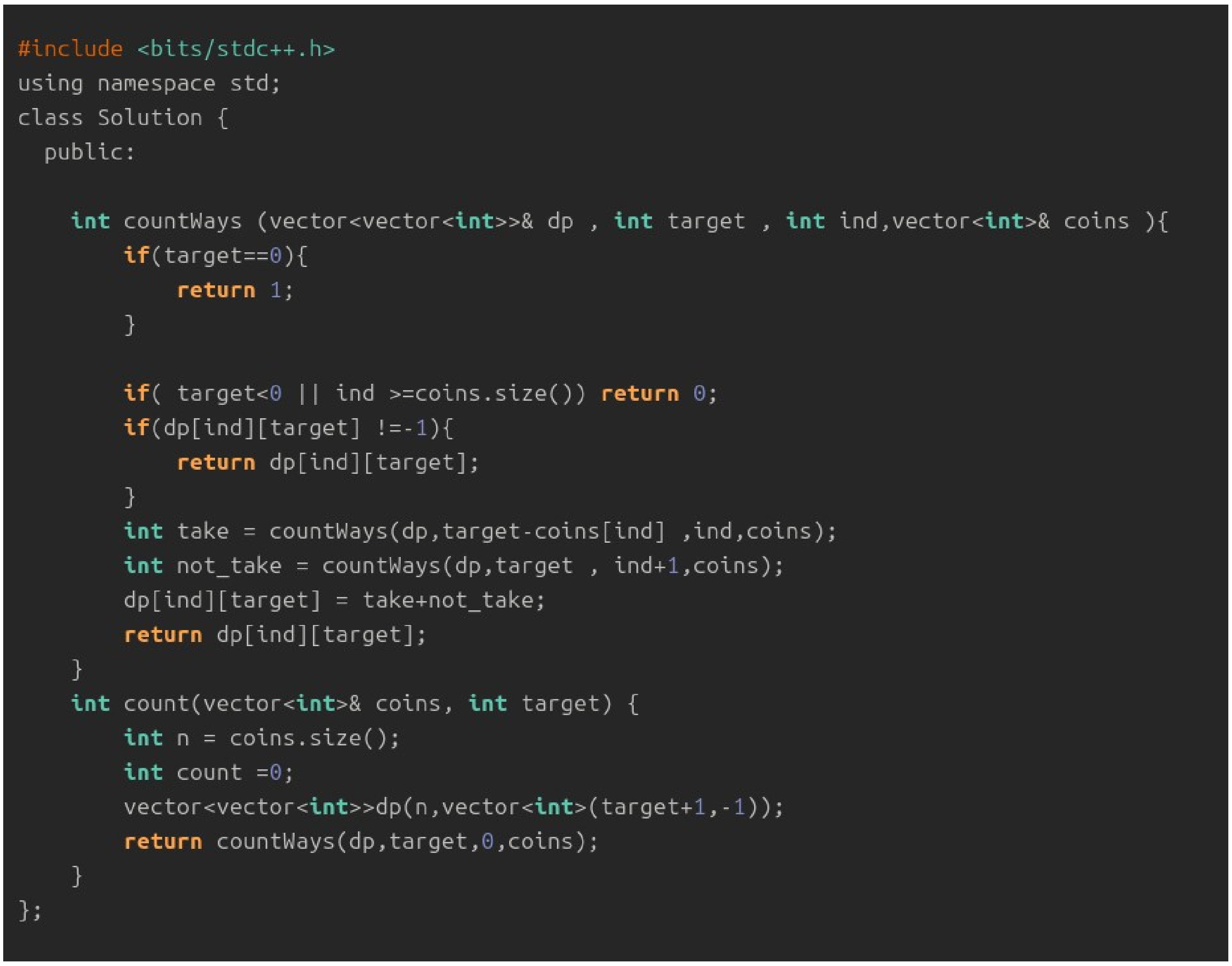
Given an integer array **coins[ ]** representing different denominations of currency and an integer **sum**, find the number of ways you can make **sum** by using different combinations from coins[ ].
Note: Assume that you have an infinite supply of each type of coin. And you can use any coin as many 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].
```

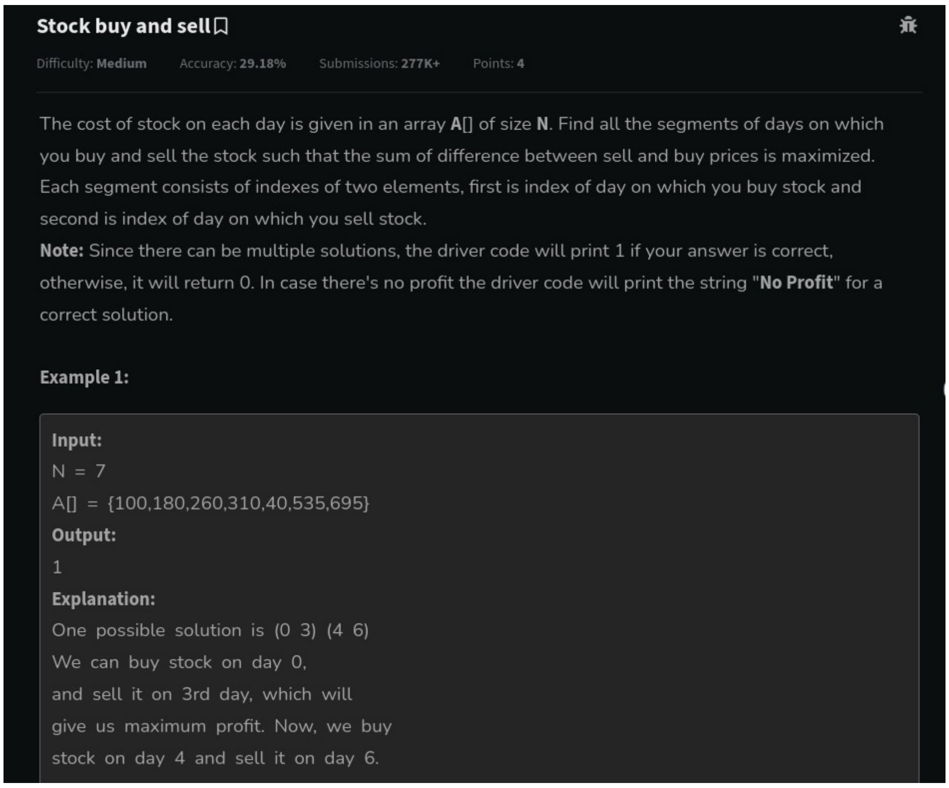
```
Input: coins[] = [5, 10], sum = 3
Output: 0
Explanation: Since all coin denominations are greater than sum, no combination can make the target sum.
```



```cpp
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:

    int countWays (vector<vector<int>>& dp , int target , int ind,vector<int>& coins ){
        if(target==0){
            return 1;
        }

        if( target<0 || ind >=coins.size()) return 0;
        if(dp[ind][target] !=-1){
            return dp[ind][target];
        }
        int take = countWays(dp,target-coins[ind] ,ind,coins);
        int not_take = countWays(dp,target , ind+1,coins);
        dp[ind][target] = take+not_take;
        return dp[ind][target];
    }
    int count(vector<int>& coins, int target) {
        int n = coins.size();
        int count =0;
        vector<vector<int>>dp(n,vector<int>(target+1,-1));
        return countWays(dp,target,0,coins);
    }
};
```

Time Complexity: O(n×target)

Space Complexity: O(n×target)



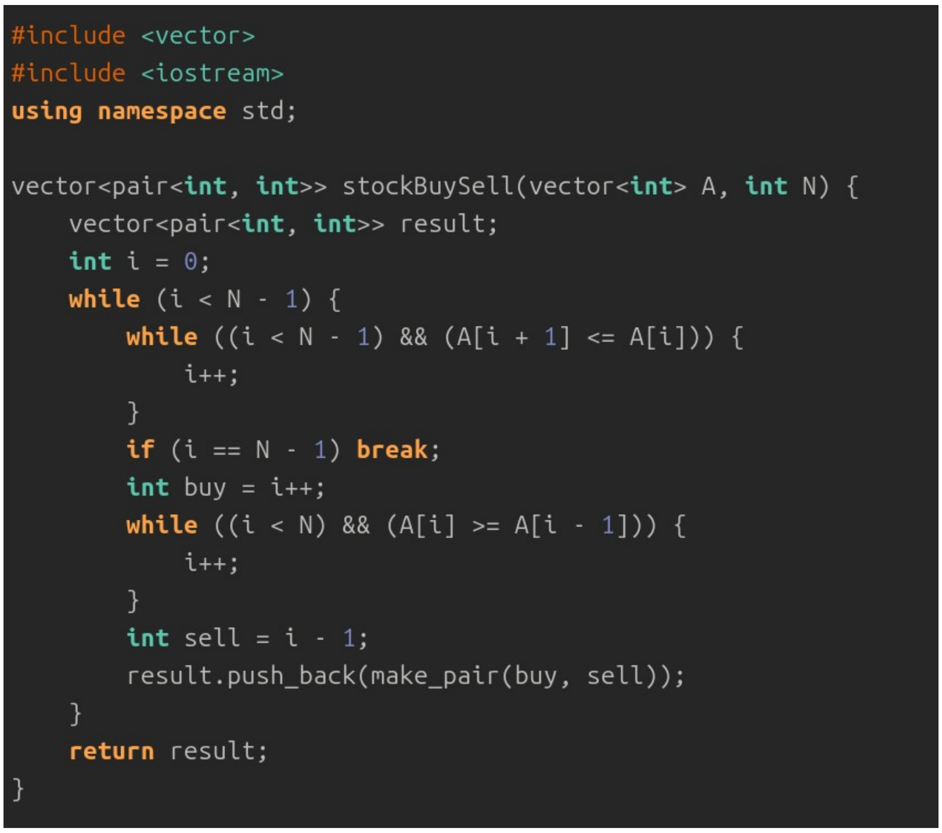
Stock buy and sell

Difficulty: **Medium** Accuracy: **29.18%** Submissions: **277K+** Points: **4**

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:

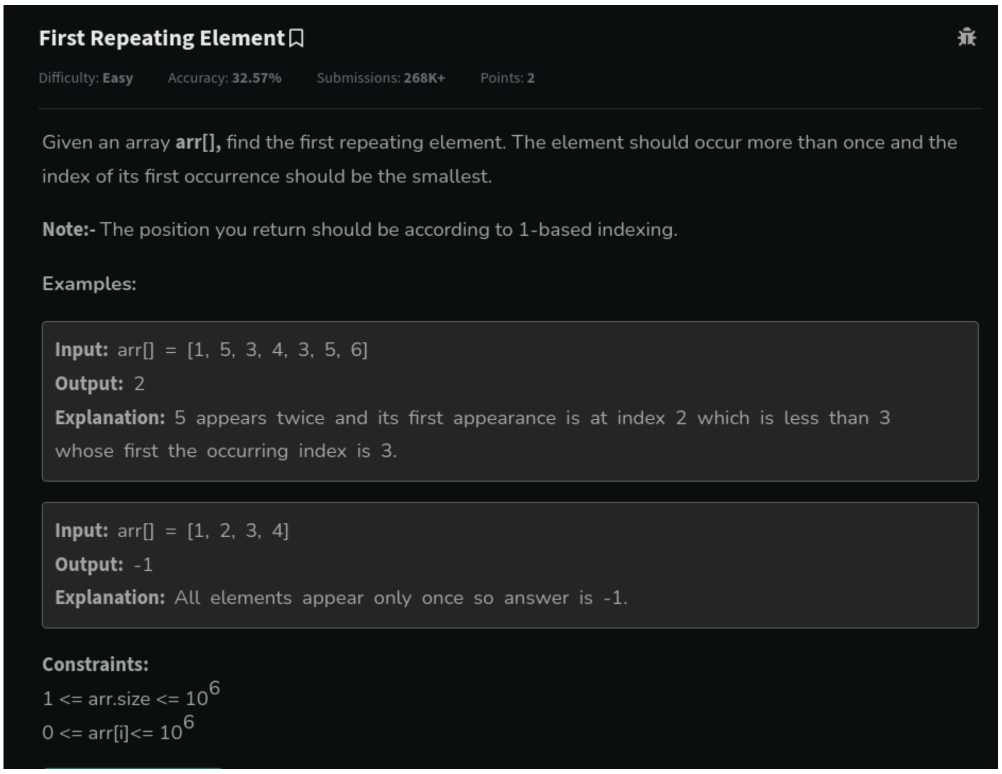
```
Input:
N = 7
A[] = {100,180,260,310,40,535,695}
Output:
1
Explanation:
One possible solution is (0 3) (4 6)
We can buy stock on day 0,
and sell it on 3rd day, which will
give us maximum profit. Now, we buy
stock on day 4 and sell it on day 6.
```



```cpp
#include <vector>
#include <iostream>
using namespace std;

vector<pair<int, int>> stockBuySell(vector<int> A, int N) {
    vector<pair<int, int>> result;
    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;
        result.push_back(make_pair(buy, sell));
    }
    return result;
}
```

Time Complexity: O(N)

Space Complexity: O(N)


First Repeating Element

Difficulty: **Easy** Accuracy: **32.57%** Submissions: **268K+** Points: **2**

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.

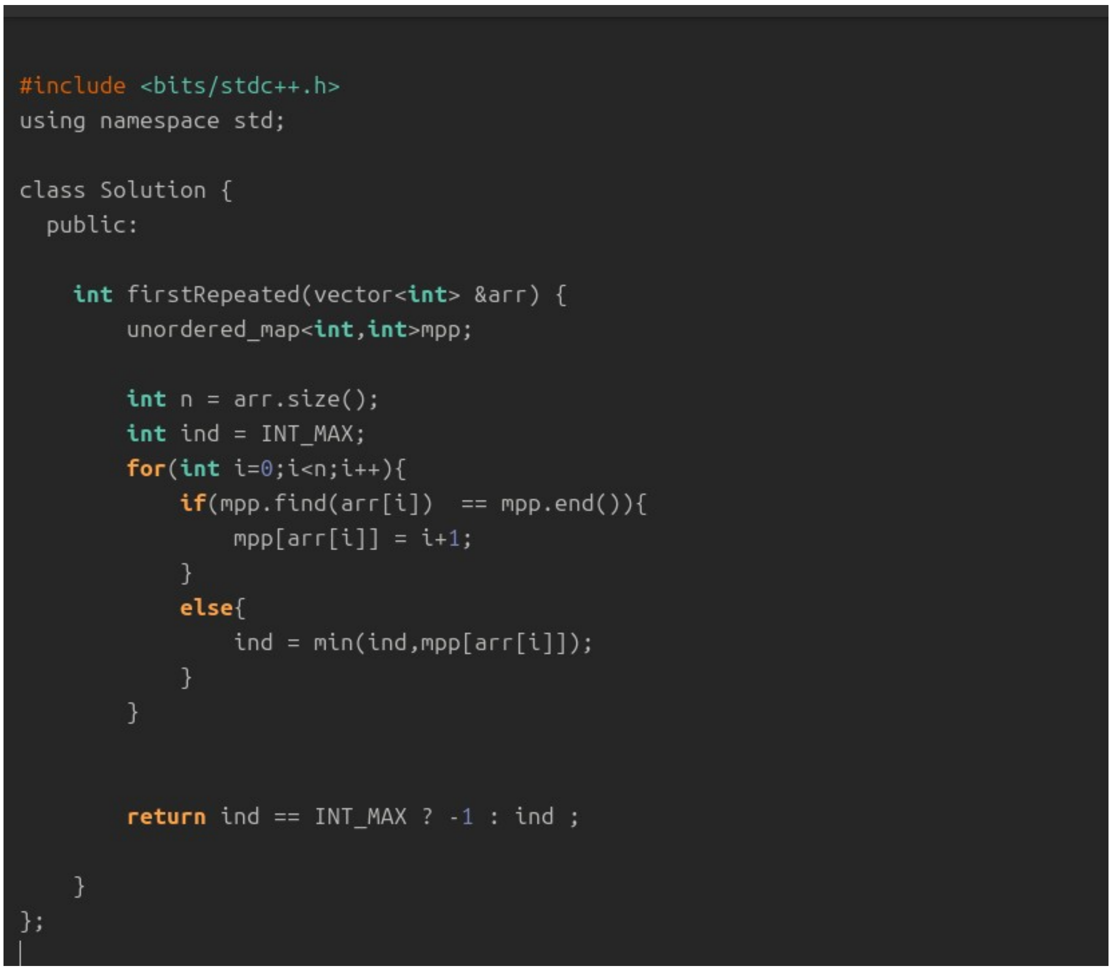
Note:- The position you return should be according to 1-based indexing.

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.
```

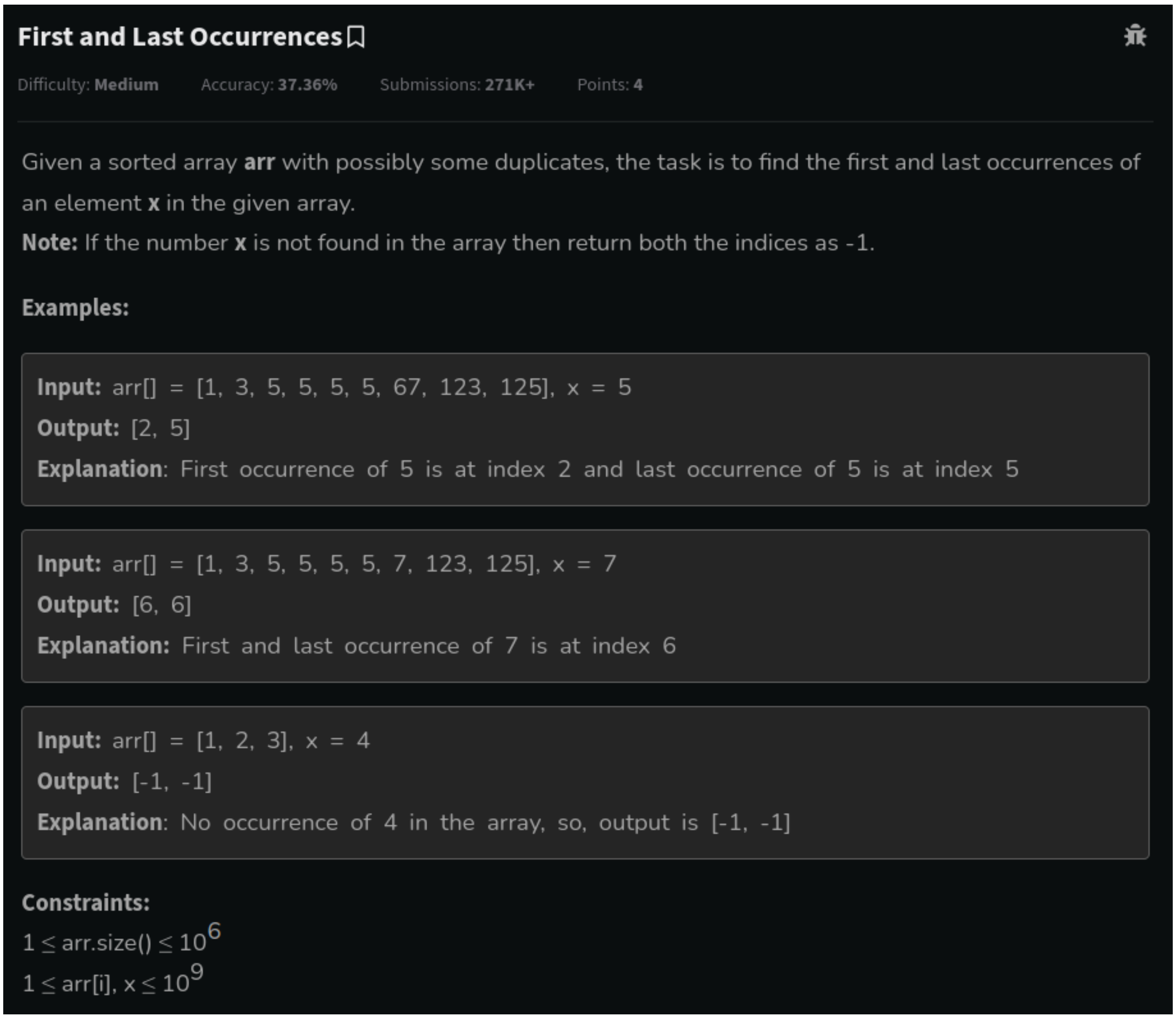
Constraints:
1 <= arr.size <= 10^6
0 <= arr[i]<= 10^6



```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
  public:

    int firstRepeated(vector<int> &arr) {
        unordered_map<int,int>mpp;

        int n = arr.size();
        int ind = INT_MAX;
        for(int i=0;i<n;i++){
            if(mpp.find(arr[i])  == mpp.end()){
                mpp[arr[i]] = i+1;
            }
            else{
                ind = min(ind,mpp[arr[i]]);
            }
        }


        return ind == INT_MAX ? -1 : ind ;

    }
};
```

Time Complexity: O(n)

Space Complexity: O(n)


First and Last Occurrences

Difficulty: **Medium** Accuracy: **37.36%** Submissions: **271K+** Points: **4**

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
```

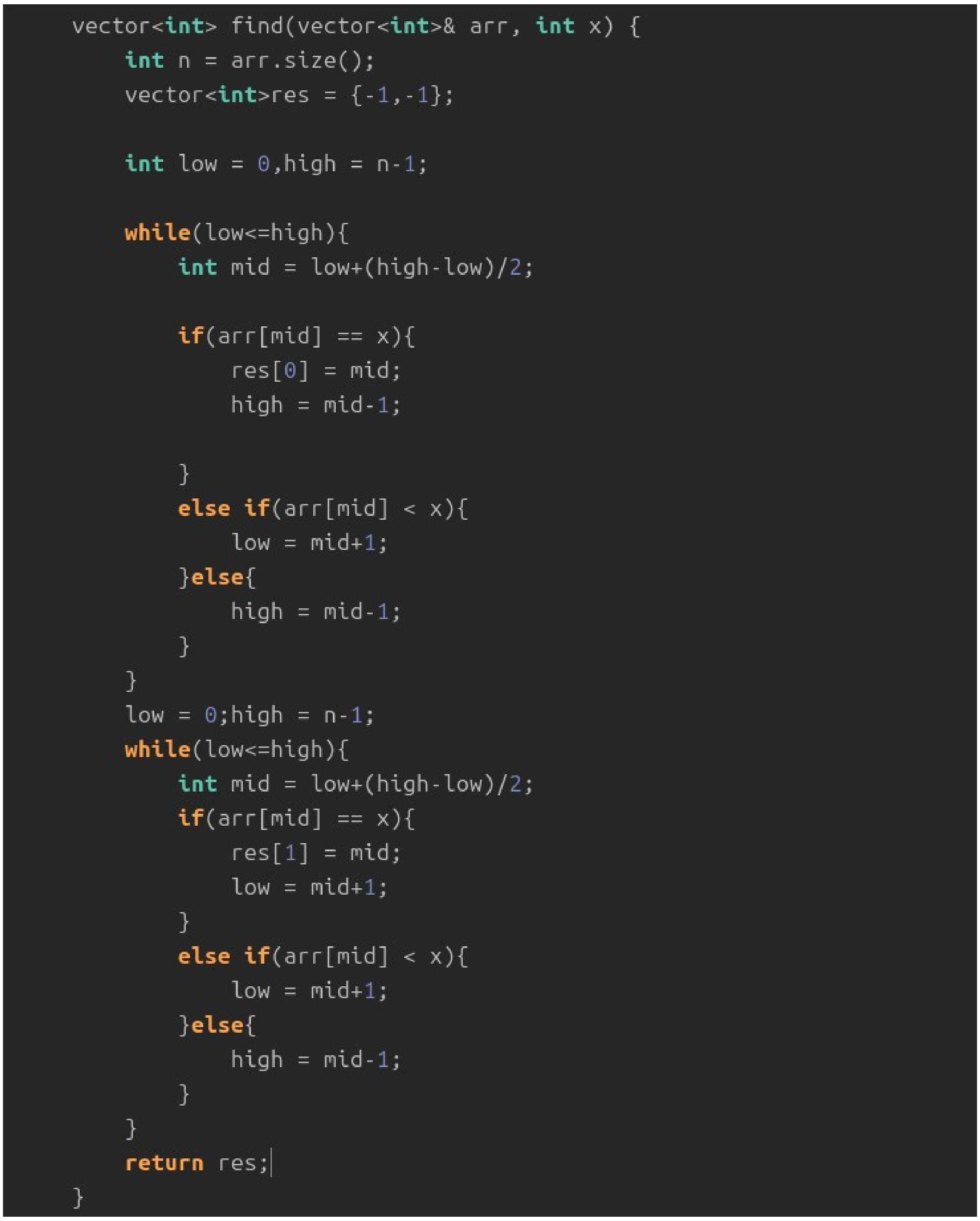
```
Input: arr[] = [1, 3, 5, 5, 5, 5, 7, 123, 125], x = 7
Output: [6, 6]
Explanation: First and last occurrence of 7 is at index 6
```

```
Input: arr[] = [1, 2, 3], x = 4
Output: [-1, -1]
Explanation: No occurrence of 4 in the array, so, output is [-1, -1]
```

Constraints:

$1 \le arr.size() \le 10^6$

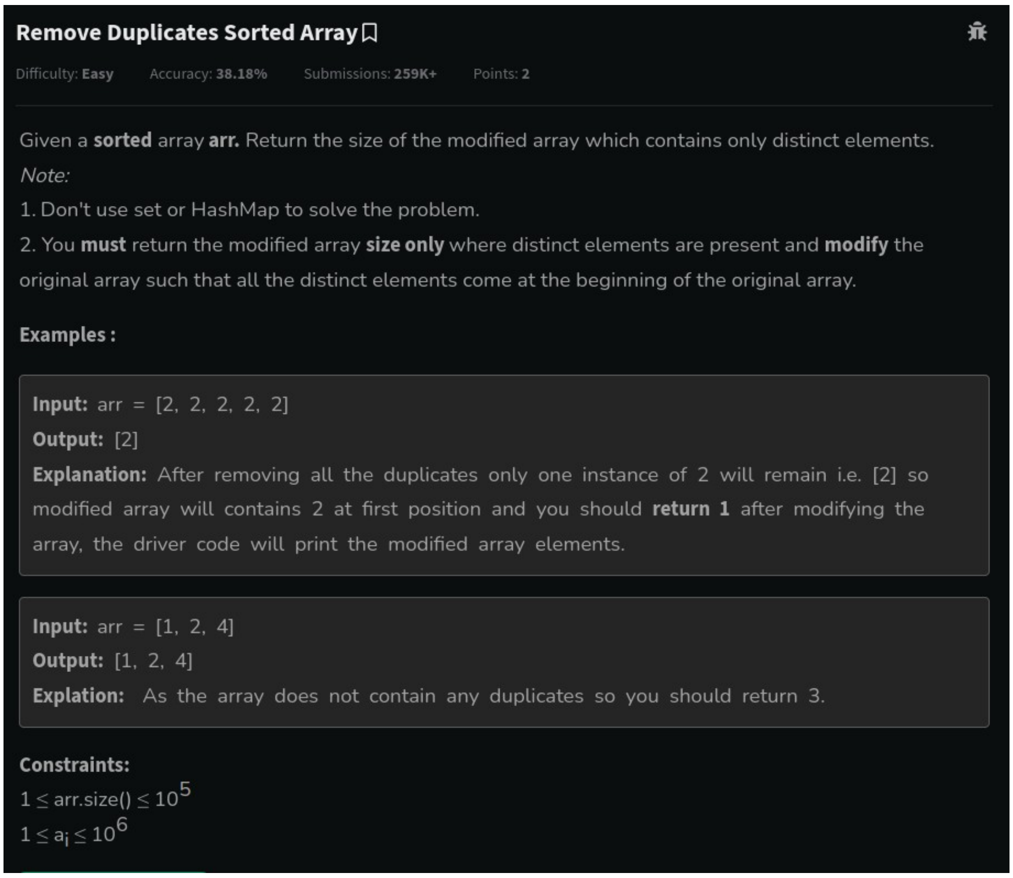
$1 \le arr[i], x \le 10^9$



```
vector<int> find(vector<int>& arr, int x) {
    int n = arr.size();
    vector<int>res = {-1,-1};

    int low = 0,high = n-1;

    while(low<=high){
        int mid = low+(high-low)/2;

        if(arr[mid] == x){
            res[0] = mid;
            high = mid-1;

        }
        else if(arr[mid] < x){
            low = mid+1;
        }else{
            high = mid-1;
        }
    }
    low = 0;high = n-1;
    while(low<=high){
        int mid = low+(high-low)/2;
        if(arr[mid] == x){
            res[1] = mid;
            low = mid+1;
        }
        else if(arr[mid] < x){
            low = mid+1;
        }else{
            high = mid-1;
        }
    }
    return res;
}
```

Time Complexity: O(logN)

space Complexity: O(1)



Remove Duplicates Sorted Array

Difficulty: **Easy** Accuracy: **38.18%** Submissions: **259K+** Points: **2**

Given a **sorted** array **arr.** Return the size of the modified array which contains only distinct elements.

Note:

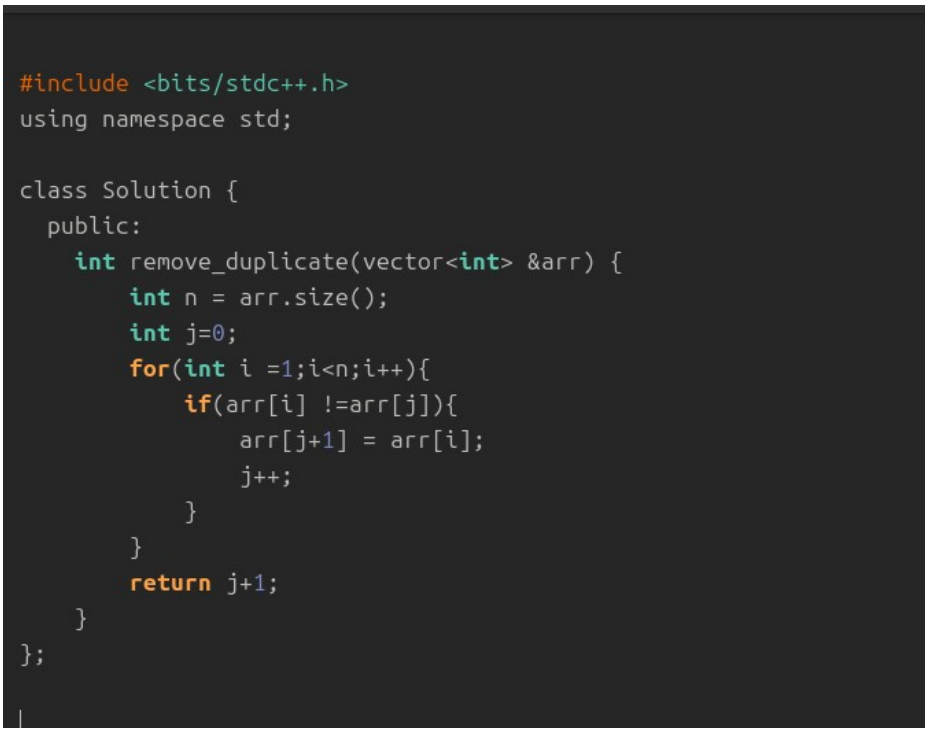
1. Don't use set or HashMap to solve the problem.
2. You **must** return the modified array **size only** where distinct elements are present and **modify** the original array such that all the distinct elements come at the beginning of the original array.

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.

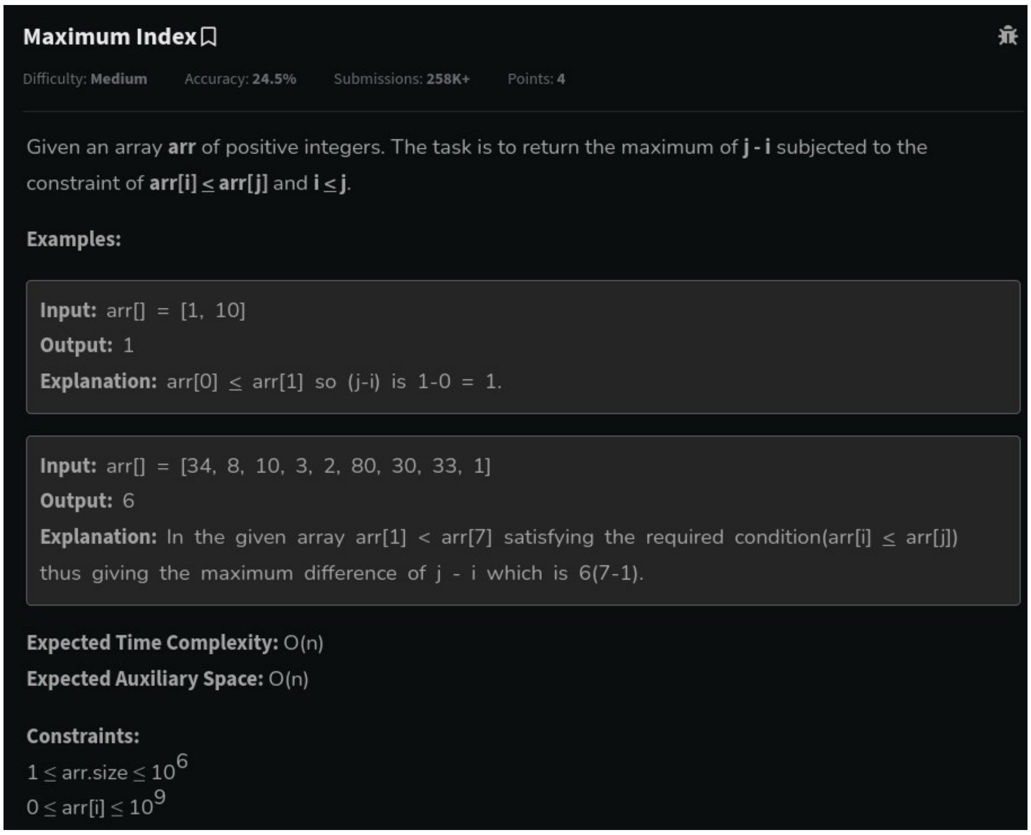
Constraints:
$1 \le arr.size() \le 10^5$
$1 \le a_i \le 10^6$



```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
  public:
    int remove_duplicate(vector<int> &arr) {
        int n = arr.size();
        int j=0;
        for(int i =1;i<n;i++){
            if(arr[i] !=arr[j]){
                arr[j+1] = arr[i];
                j++;
            }
        }
        return j+1;
    }
};
```

Time Complexity : O(N)

Space Complexity:O(1)



Maximum Index

Difficulty: **Medium** Accuracy: **24.5%** Submissions: **258K+** Points: **4**

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**.

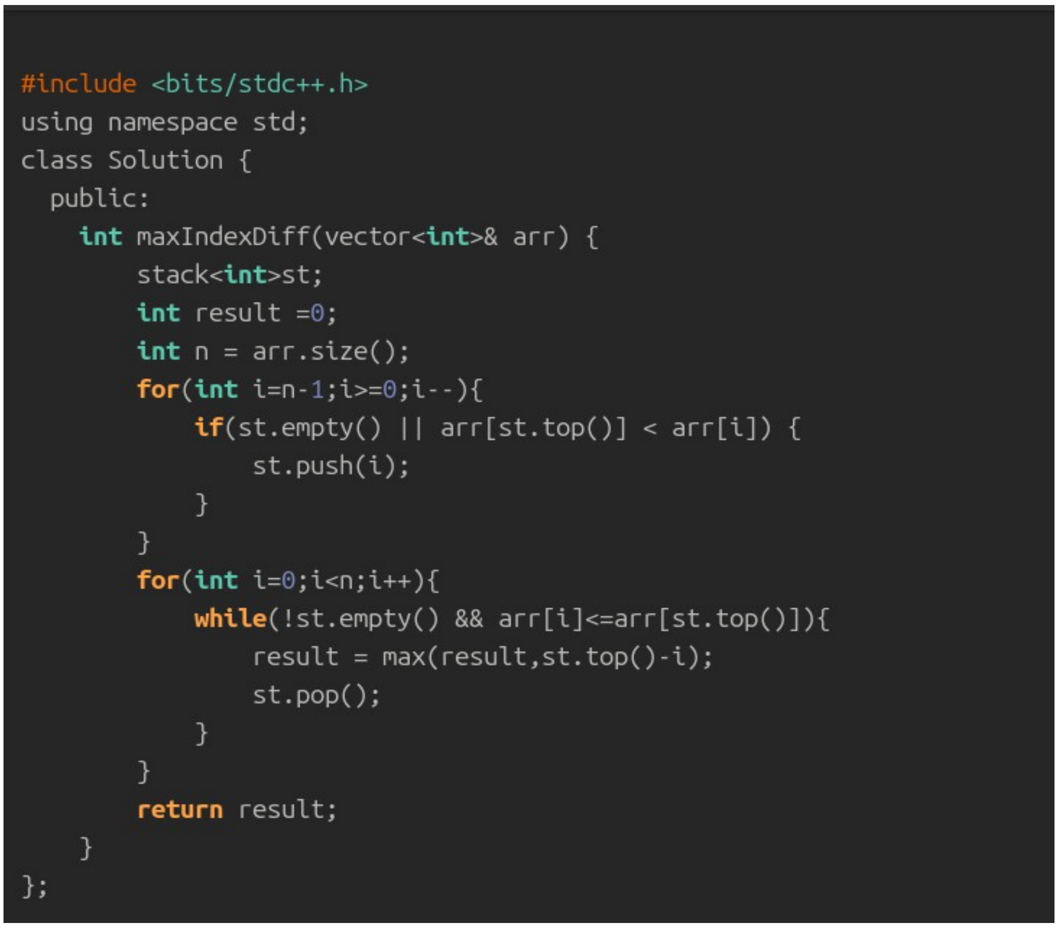
Examples:

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)

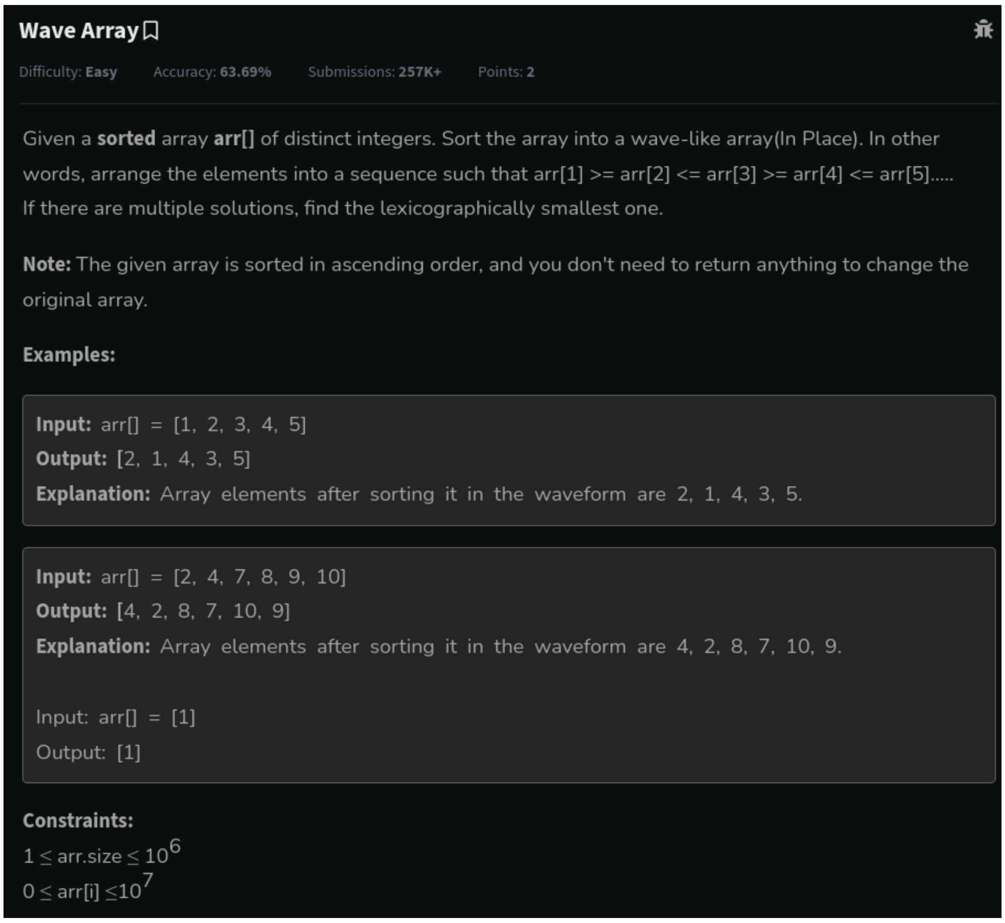
Constraints:
$1 \le arr.size \le 10^6$
$0 \le arr[i] \le 10^9$



```cpp
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
    int maxIndexDiff(vector<int>& arr) {
        stack<int>st;
        int result =0;
        int n = arr.size();
        for(int i=n-1;i>=0;i--){
            if(st.empty() || arr[st.top()] < arr[i]) {
                st.push(i);
            }
        }
        for(int i=0;i<n;i++){
            while(!st.empty() && arr[i]<=arr[st.top()]){
                result = max(result,st.top()-i);
                st.pop();
            }
        }
        return result;
    }
};
```

Time Complexity : O(N)

Space Complexity: O(N)



Wave Array

Difficulty: **Easy** Accuracy: **63.69%** Submissions: **257K+** Points: **2**

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.

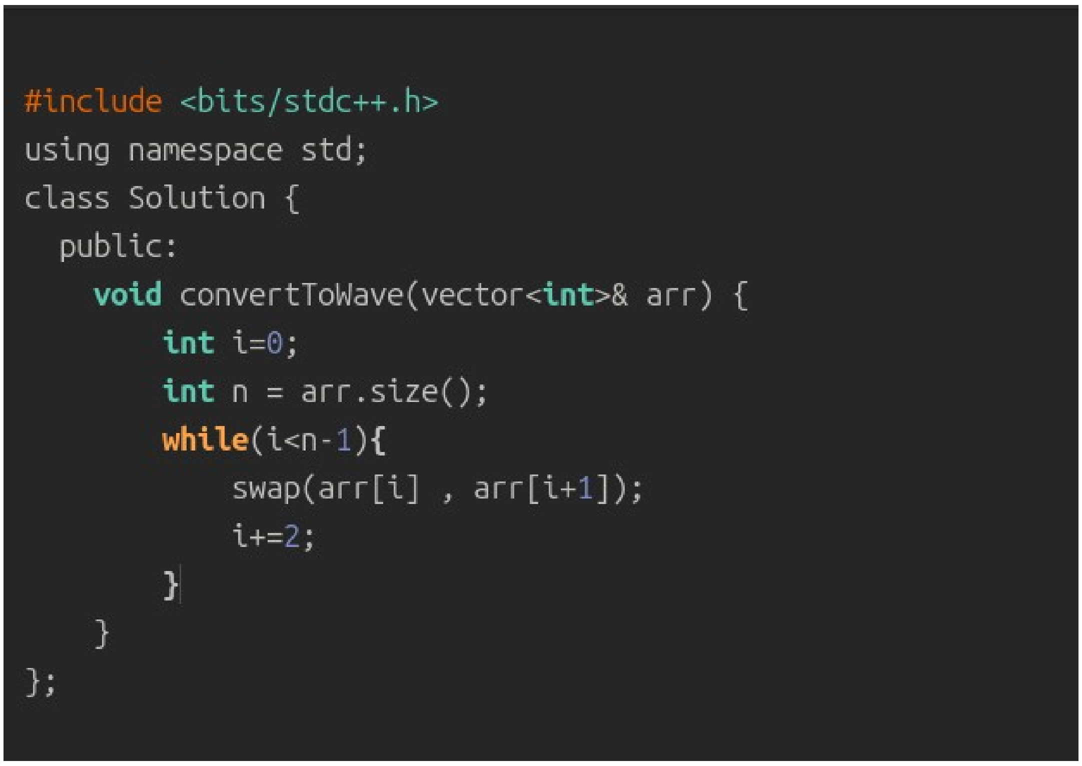
Note: The given array is sorted in ascending order, and you don't need to return anything to change the original array.

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]
```

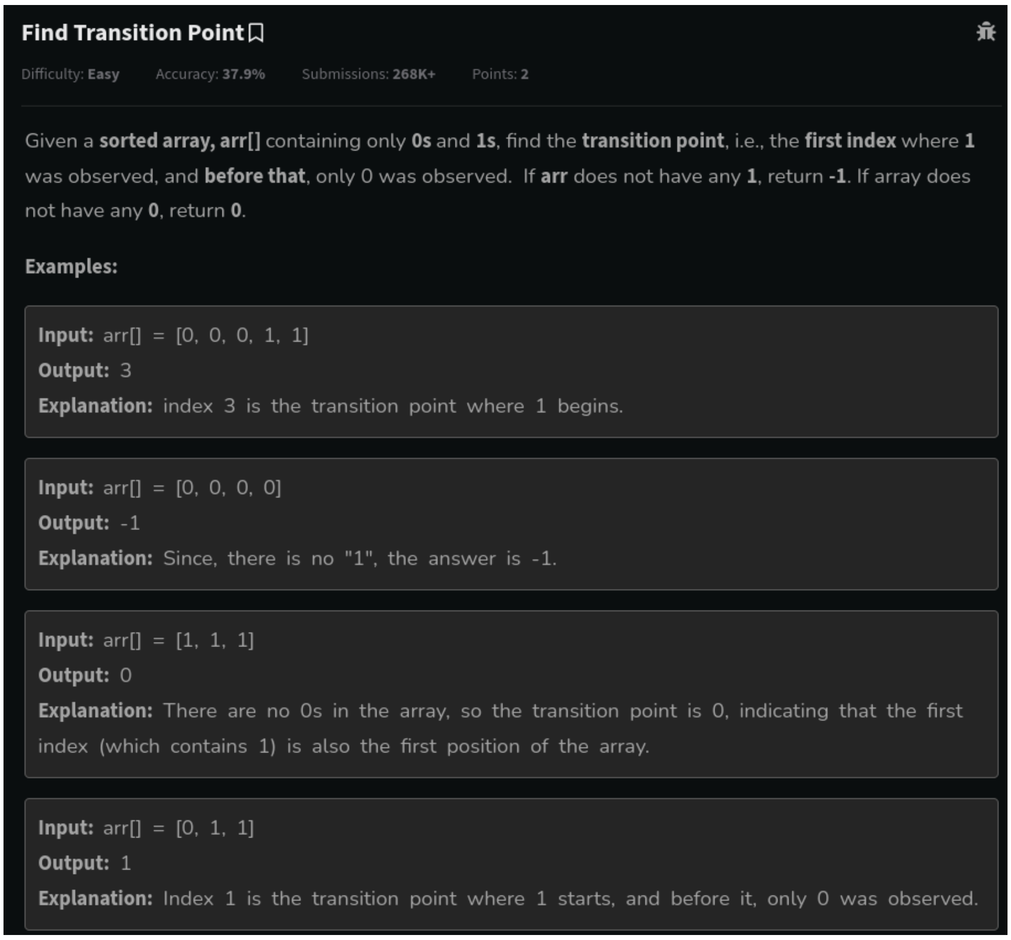
Constraints:
$1 \le arr.size \le 10^6$
$0 \le arr[i] \le 10^7$



```cpp
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
    void convertToWave(vector<int>& arr) {
        int i=0;
        int n = arr.size();
        while(i<n-1){
            swap(arr[i] , arr[i+1]);
            i+=2;
        }
    }
};
```

Time Complexity : O(n)

Space Complexity : O(1)



Find Transition Point

Difficulty: **Easy** Accuracy: **37.9%** Submissions: **268K+** Points: **2**

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**.

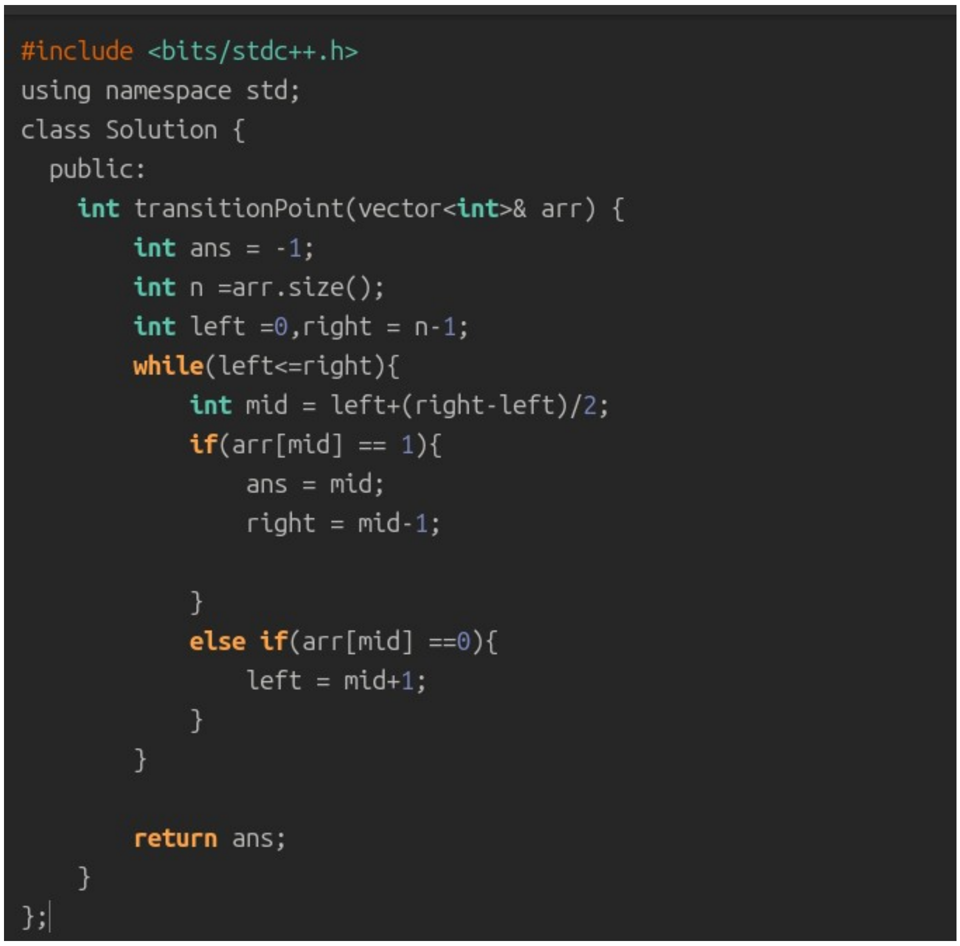
Examples:

```
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.
```



```cpp
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
    int transitionPoint(vector<int>& arr) {
        int ans = -1;
        int n =arr.size();
        int left =0,right = n-1;
        while(left<=right){
            int mid = left+(right-left)/2;
            if(arr[mid] == 1){
                ans = mid;
                right = mid-1;

            }
            else if(arr[mid] ==0){
                left = mid+1;
            }
        }

        return ans;
    }
};
```

Time Complexity : O(logN)   Space Complexity : O(1)