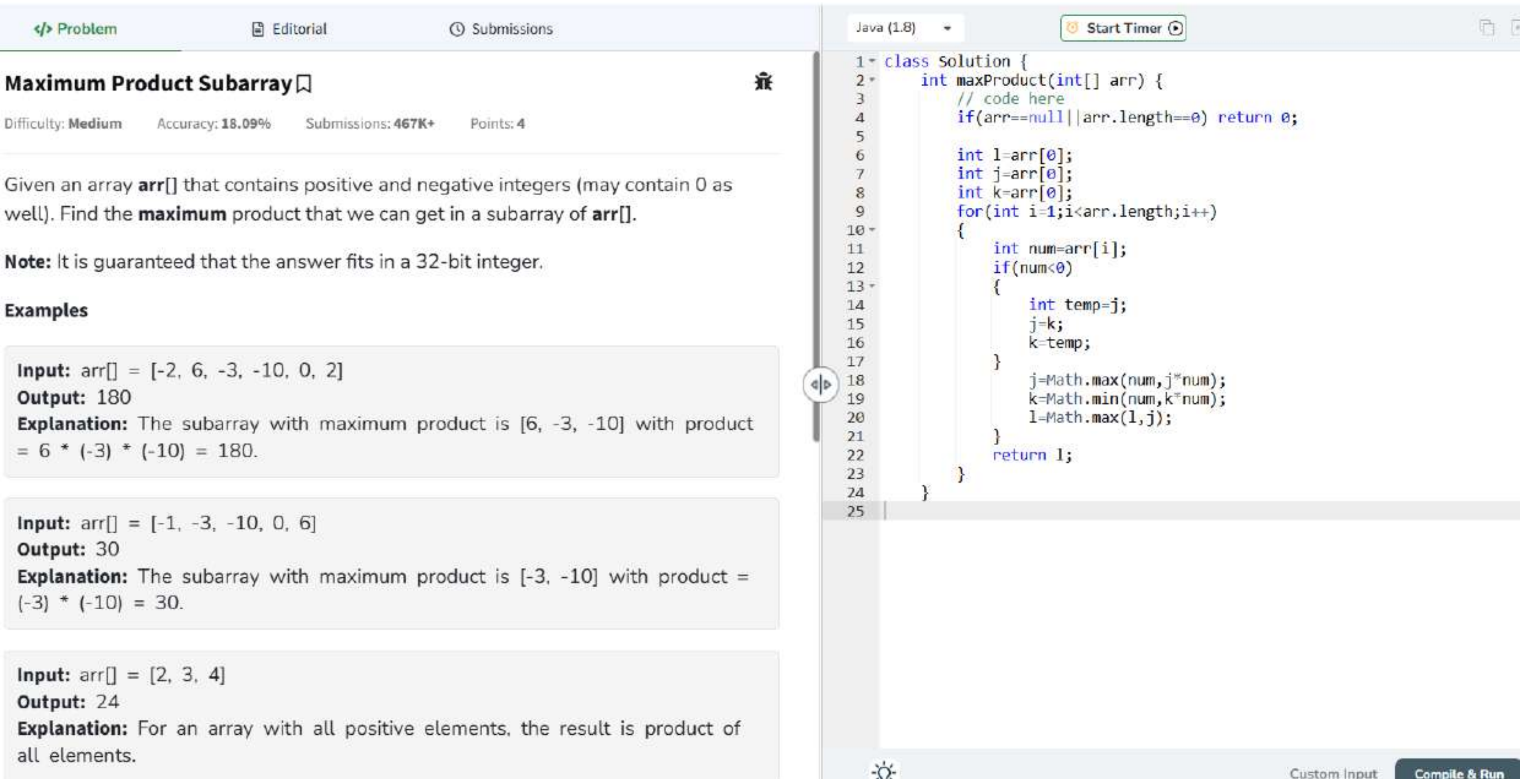


Maximum Product Subarray

Difficulty: **Medium** Accuracy: **18.09%** Submissions: **467K+** Points: **4**

Given an array **arr[]** that contains positive and negative integers (may contain 0 as well). Find the **maximum** product that we can get in a subarray of **arr[]**.

Note: It is guaranteed that the answer fits in a 32-bit integer.

Examples

Input: arr[] = [-2, 6, -3, -10, 0, 2]
Output: 180
Explanation: The subarray with maximum product is [6, -3, -10] with product = 6 * (-3) * (-10) = 180.

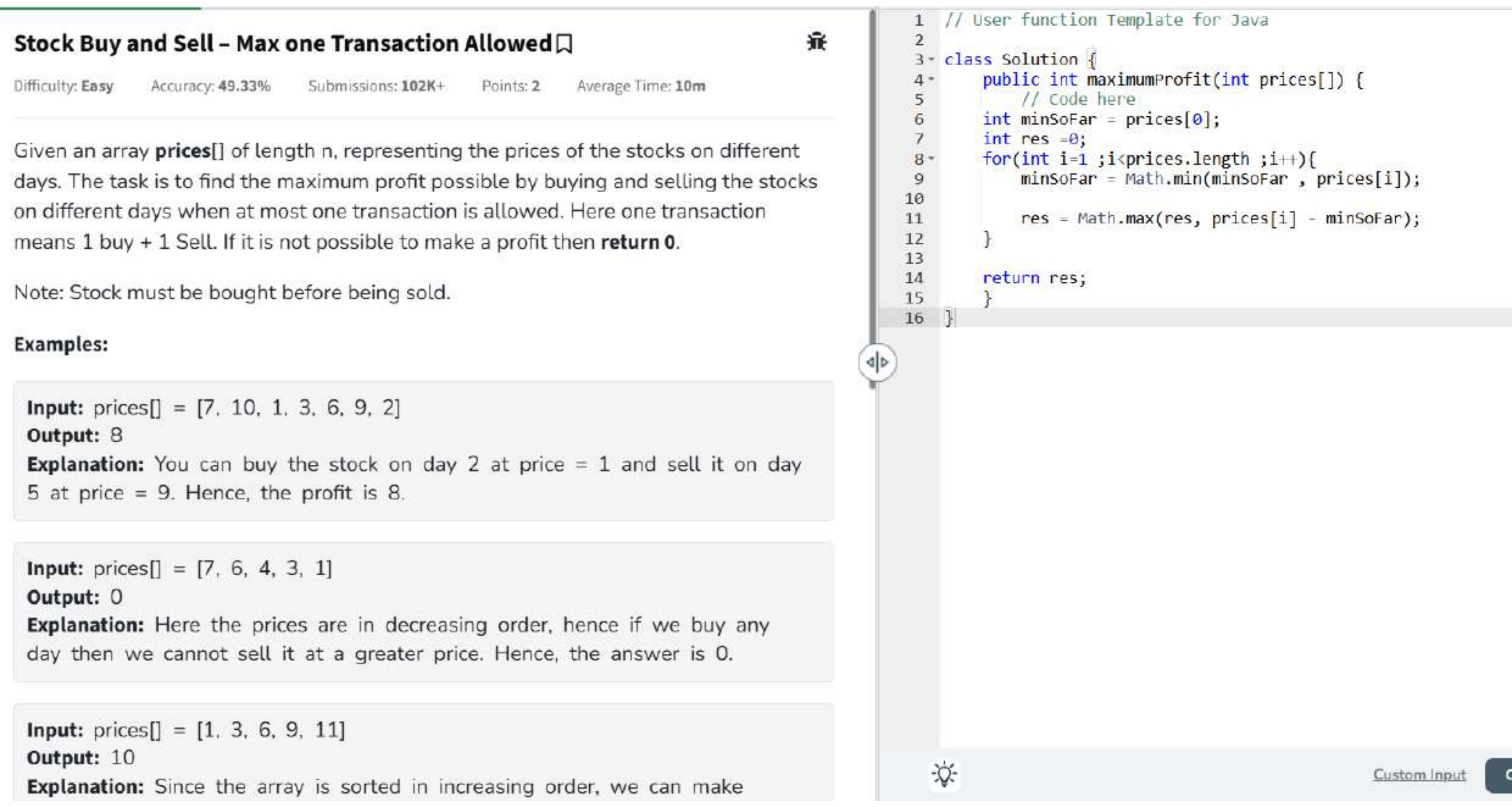
Input: arr[] = [-1, -3, -10, 0, 6]
Output: 30
Explanation: The subarray with maximum product is [-3, -10] with product = (-3) * (-10) = 30.

Input: arr[] = [2, 3, 4]
Output: 24
Explanation: For an array with all positive elements, the result is product of all elements.

```java
class Solution {
    int maxProduct(int[] arr) {
        // code here
        if(arr==null||arr.length==0) return 0;

        int l=arr[0];
        int j=arr[0];
        int k=arr[0];
        for(int i=1;i<arr.length;i++)
        {
            int num=arr[i];
            if(num<0)
            {
                int temp=j;
                j=k;
                k=temp;
            }
                j=Math.max(num,j*num);
                k=Math.min(num,k*num);
                l=Math.max(l,j);
            }
            return l;
        }
    }

```



Stock Buy and Sell – Max one Transaction Allowed

Difficulty: **Easy** Accuracy: **49.33%** Submissions: **102K+** Points: **2** Average Time: **10m**

Given an array **prices**[] of length n, representing the prices of the stocks on different days. The task is to find the maximum profit possible by buying and selling the stocks on different days when at most one transaction is allowed. Here one transaction means 1 buy + 1 Sell. If it is not possible to make a profit then **return 0**.

Note: Stock must be bought before being sold.

Examples:

Input: prices[] = [7, 10, 1, 3, 6, 9, 2]
Output: 8
Explanation: You can buy the stock on day 2 at price = 1 and sell it on day 5 at price = 9. Hence, the profit is 8.

Input: prices[] = [7, 6, 4, 3, 1]
Output: 0
Explanation: Here the prices are in decreasing order, hence if we buy any day then we cannot sell it at a greater price. Hence, the answer is 0.

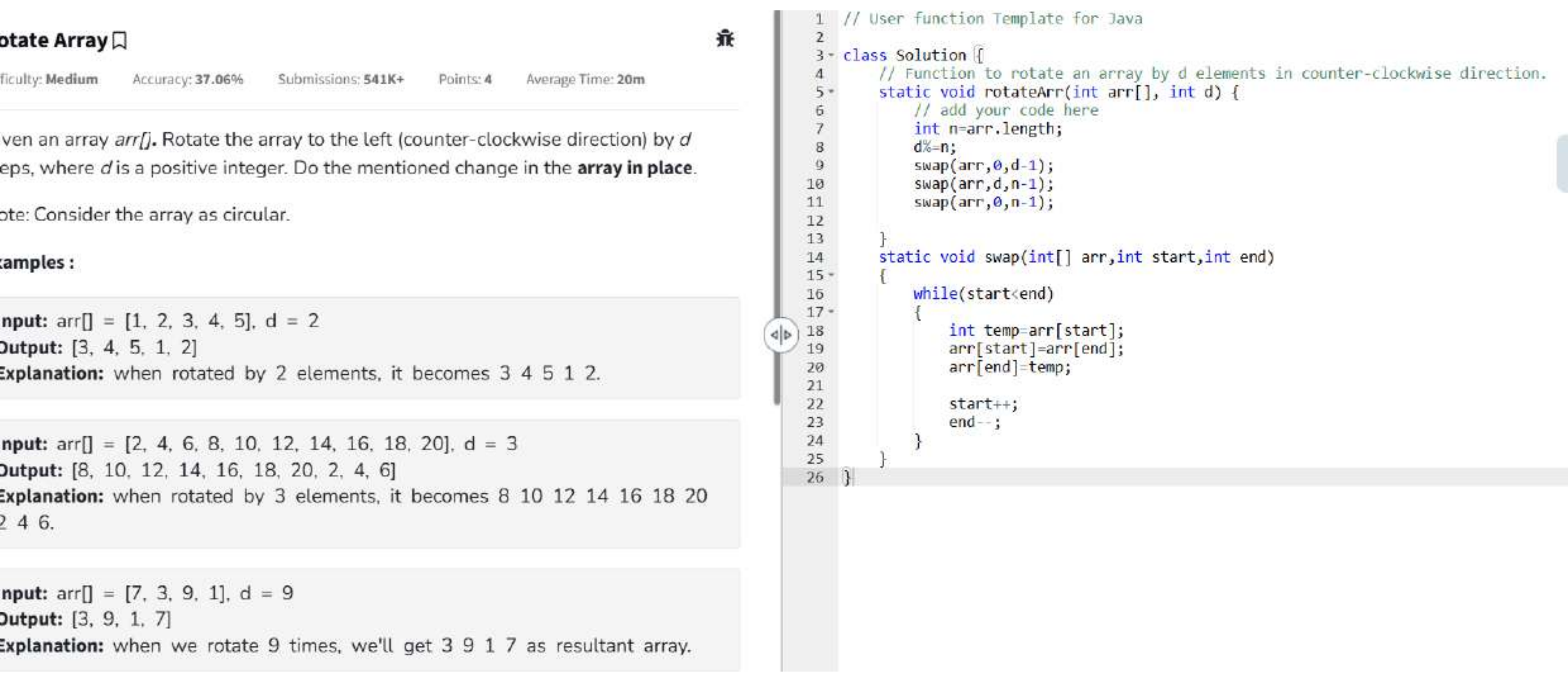
Input: prices[] = [1, 3, 6, 9, 11]
Output: 10
Explanation: Since the array is sorted in increasing order, we can make

```java
// User function Template for Java

class Solution {
    public int maximumProfit(int prices[]) {
        // Code here
    int minSoFar = prices[0];
    int res =0;
    for(int i=1 ;i<prices.length ;i++){
        minSoFar = Math.min(minSoFar , prices[i]);

        res = Math.max(res, prices[i] - minSoFar);
    }

    return res;
    }
}
```



otate Array

ficulty: **Medium** Accuracy: **37.06%** Submissions: **541K+** Points: **4** Average Time: **20m**

ven an array *arr[]*. Rotate the array to the left (counter-clockwise direction) by *d*
eps, where *d* is a positive integer. Do the mentioned change in the **array in place**.

ote: Consider the array as circular.

amples :

nput: arr[] = [1, 2, 3, 4, 5], d = 2
Output: [3, 4, 5, 1, 2]
Explanation: when rotated by 2 elements, it becomes 3 4 5 1 2.

nput: arr[] = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20], d = 3
Output: [8, 10, 12, 14, 16, 18, 20, 2, 4, 6]
Explanation: when rotated by 3 elements, it becomes 8 10 12 14 16 18 20
2 4 6.

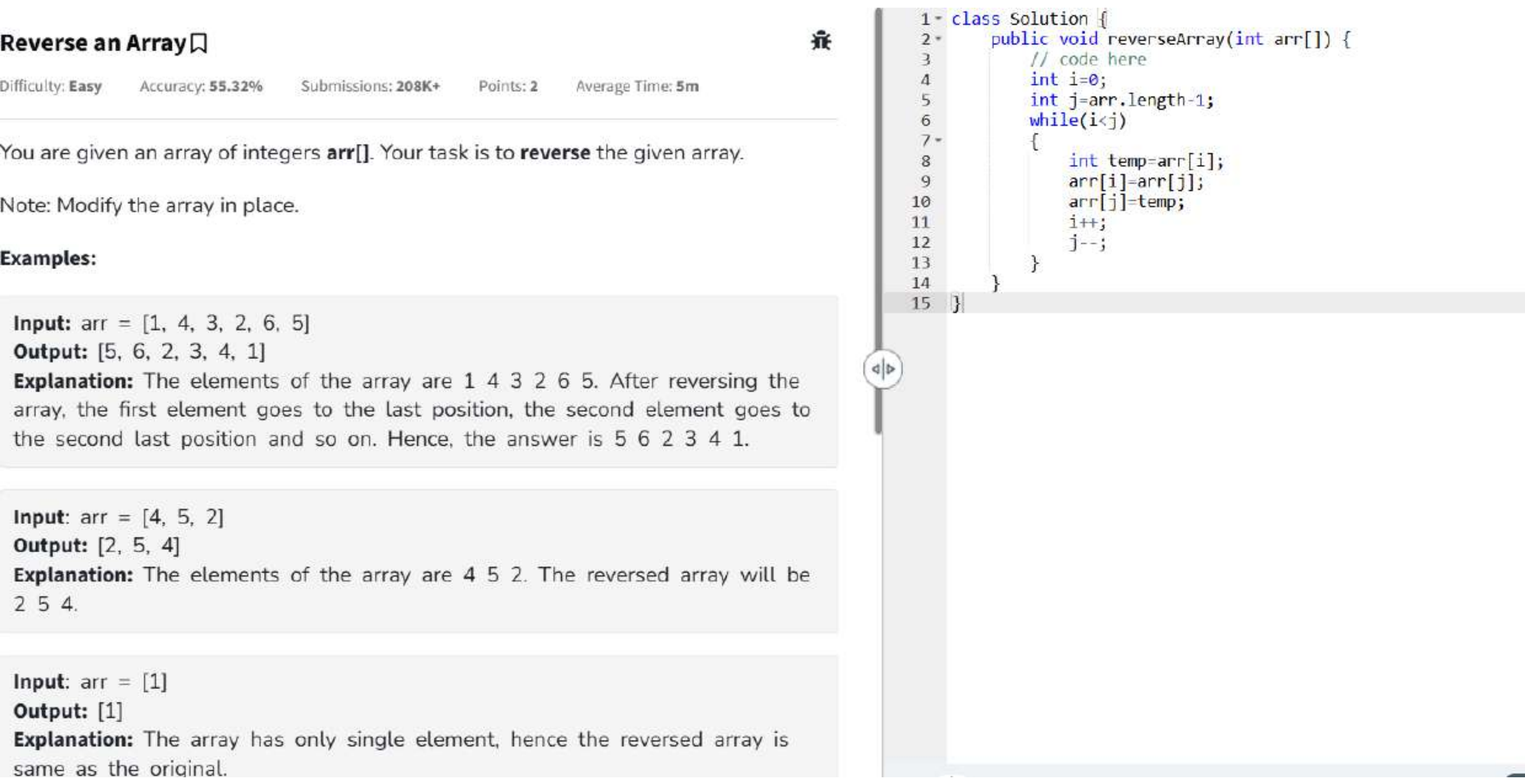
nput: arr[] = [7, 3, 9, 1], d = 9
Output: [3, 9, 1, 7]
Explanation: when we rotate 9 times, we'll get 3 9 1 7 as resultant array.

```java
// User function Template for Java

class Solution {
    // Function to rotate an array by d elements in counter-clockwise direction.
    static void rotateArr(int arr[], int d) {
        // add your code here
        int n=arr.length;
        d%=n;
        swap(arr,0,d-1);
        swap(arr,d,n-1);
        swap(arr,0,n-1);

    }
    static void swap(int[] arr,int start,int end)
    {
        while(start<end)
        {
            int temp=arr[start];
            arr[start]=arr[end];
            arr[end]=temp;

            start++;
            end--;
        }
    }
}
```



Reverse an Array

Difficulty: **Easy** Accuracy: **55.32%** Submissions: **208K+** Points: **2** Average Time: **5m**

You are given an array of integers **arr[]**. Your task is to **reverse** the given array.

Note: Modify the array in place.

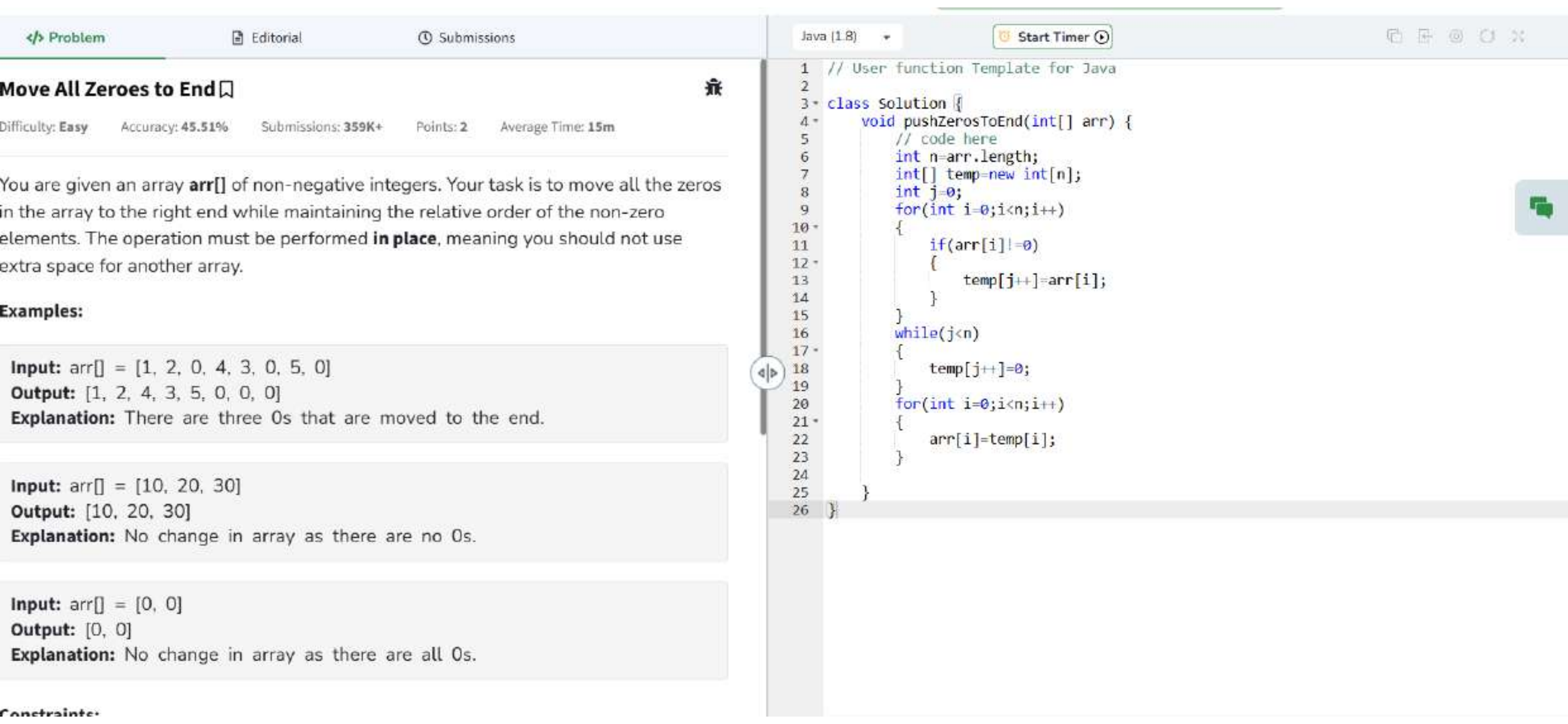
Examples:

Input: arr = [1, 4, 3, 2, 6, 5]
Output: [5, 6, 2, 3, 4, 1]
Explanation: The elements of the array are 1 4 3 2 6 5. After reversing the array, the first element goes to the last position, the second element goes to the second last position and so on. Hence, the answer is 5 6 2 3 4 1.

Input: arr = [4, 5, 2]
Output: [2, 5, 4]
Explanation: The elements of the array are 4 5 2. The reversed array will be 2 5 4.

Input: arr = [1]
Output: [1]
Explanation: The array has only single element, hence the reversed array is same as the original.

```java
class Solution {
    public void reverseArray(int arr[]) {
        // code here
        int i=0;
        int j=arr.length-1;
        while(i<j)
        {
            int temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
            i++;
            j--;
        }
    }
}
```



Move All Zeroes to End

Difficulty: **Easy** Accuracy: **45.51%** Submissions: **359K+** Points: **2** Average Time: **15m**

You are given an array **arr[]** of non-negative integers. Your task is to move all the zeros in the array to the right end while maintaining the relative order of the non-zero elements. The operation must be performed **in place**, meaning you should not use extra space for another array.

Examples:

Input: arr[] = [1, 2, 0, 4, 3, 0, 5, 0]
Output: [1, 2, 4, 3, 5, 0, 0, 0]
Explanation: There are three 0s that are moved to the end.

Input: arr[] = [10, 20, 30]
Output: [10, 20, 30]
Explanation: No change in array as there are no 0s.

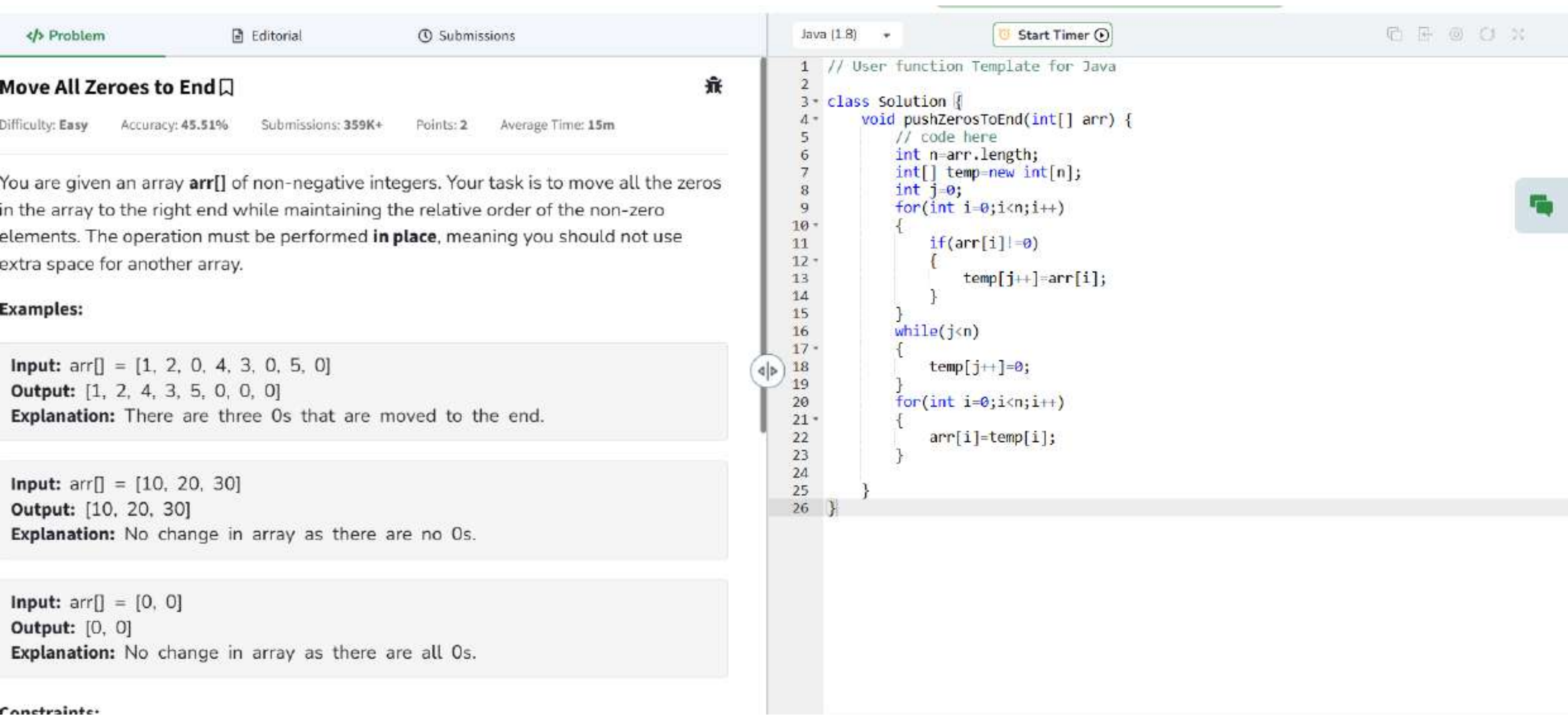
Input: arr[] = [0, 0]
Output: [0, 0]
Explanation: No change in array as there are all 0s.

Constraints:

```java
// User function Template for Java

class Solution {
    void pushZerosToEnd(int[] arr) {
        // code here
        int n=arr.length;
        int[] temp=new int[n];
        int j=0;
        for(int i=0;i<n;i++)
        {
            if(arr[i]!=0)
            {
                temp[j++]=arr[i];
            }
        }
        while(j<n)
        {
            temp[j++]=0;
        }
        for(int i=0;i<n;i++)
        {
            arr[i]=temp[i];
        }

    }
}
```



Move All Zeroes to End

Difficulty: **Easy** Accuracy: **45.51%** Submissions: **359K+** Points: **2** Average Time: **15m**

You are given an array **arr[]** of non-negative integers. Your task is to move all the zeros in the array to the right end while maintaining the relative order of the non-zero elements. The operation must be performed **in place**, meaning you should not use extra space for another array.

Examples:

Input: arr[] = [1, 2, 0, 4, 3, 0, 5, 0]
Output: [1, 2, 4, 3, 5, 0, 0, 0]
Explanation: There are three 0s that are moved to the end.

Input: arr[] = [10, 20, 30]
Output: [10, 20, 30]
Explanation: No change in array as there are no 0s.

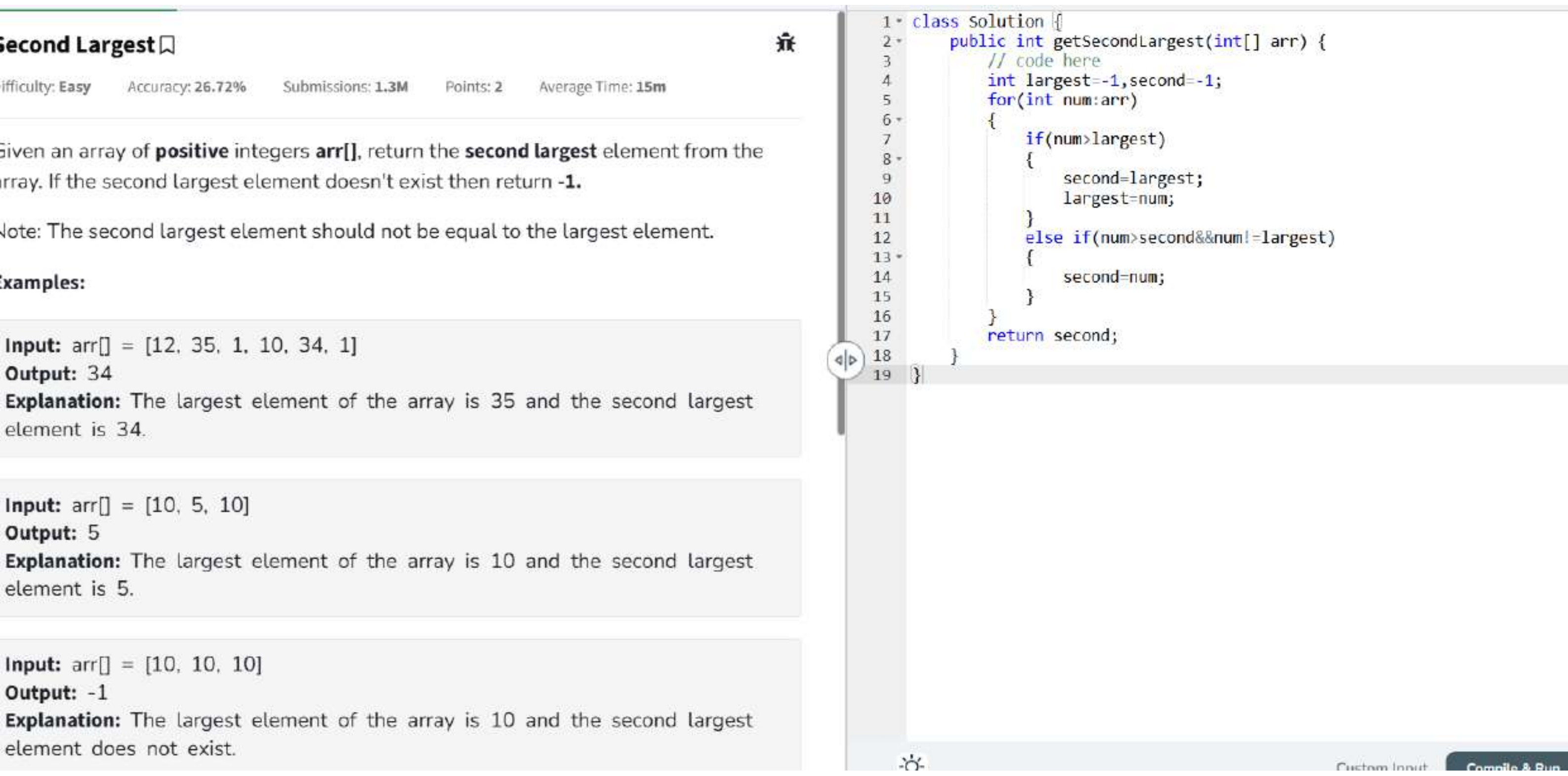
Input: arr[] = [0, 0]
Output: [0, 0]
Explanation: No change in array as there are all 0s.

Constraints:

```java
// User function Template for Java

class Solution {
    void pushZerosToEnd(int[] arr) {
        // code here
        int n=arr.length;
        int[] temp=new int[n];
        int j=0;
        for(int i=0;i<n;i++)
        {
            if(arr[i]!=0)
            {
                temp[j++]=arr[i];
            }
        }
        while(j<n)
        {
            temp[j++]=0;
        }
        for(int i=0;i<n;i++)
        {
            arr[i]=temp[i];
        }

    }
}
```



Second Largest

Difficulty: **Easy** Accuracy: **26.72%** Submissions: **1.3M** Points: **2** Average Time: **15m**

Given an array of **positive** integers **arr[]**, return the **second largest** element from the array. If the second largest element doesn't exist then return **-1.**

Note: The second largest element should not be equal to the largest element.

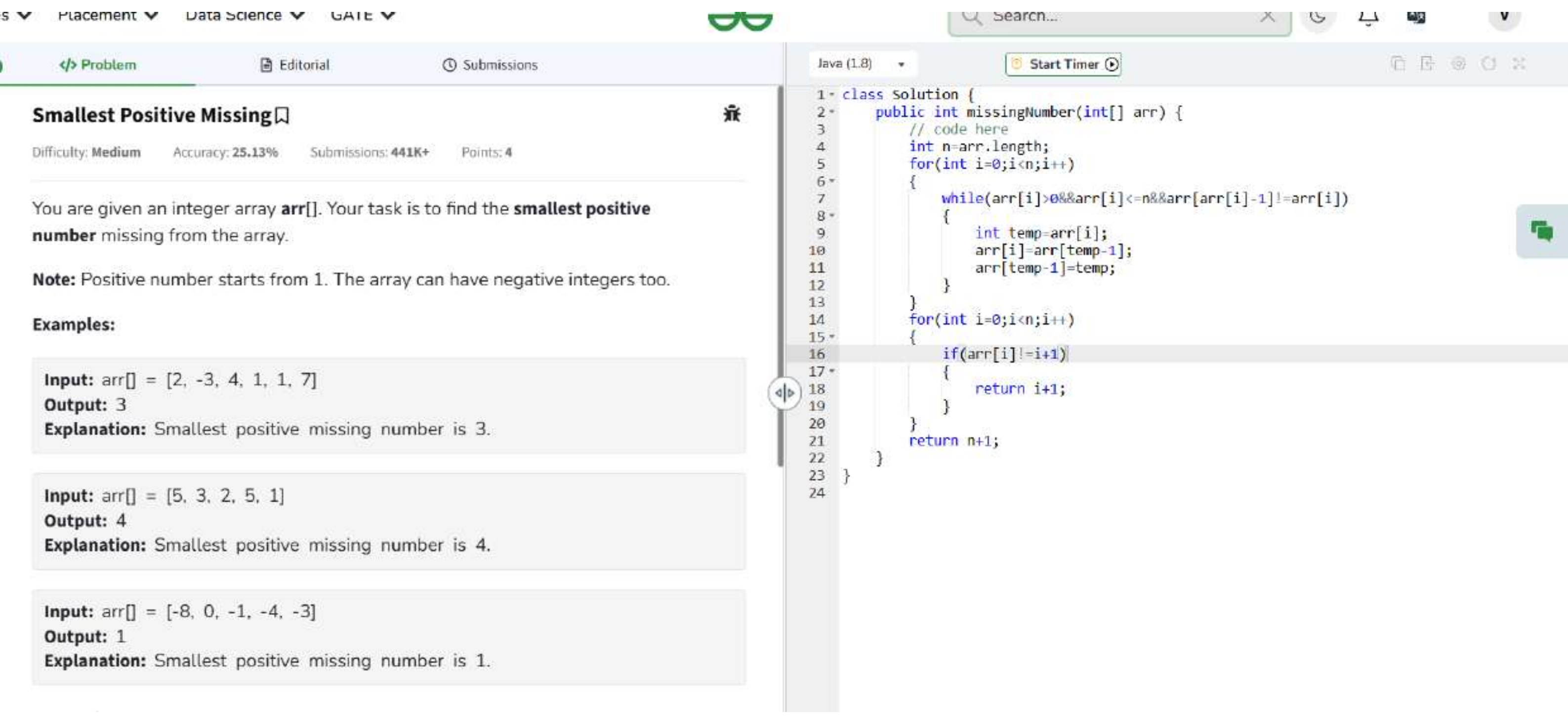
Examples:

Input: arr[] = [12, 35, 1, 10, 34, 1]
Output: 34
Explanation: The largest element of the array is 35 and the second largest element is 34.

Input: arr[] = [10, 5, 10]
Output: 5
Explanation: The largest element of the array is 10 and the second largest element is 5.

Input: arr[] = [10, 10, 10]
Output: -1
Explanation: The largest element of the array is 10 and the second largest element does not exist.

```java
class Solution {
    public int getSecondLargest(int[] arr) {
        // code here
        int largest=-1,second=-1;
        for(int num:arr)
        {
            if(num>largest)
            {
                second=largest;
                largest=num;
            }
            else if(num>second&&num!=largest)
            {
                second=num;
            }
        }
        return second;
    }
}
```



Smallest Positive Missing

Difficulty: **Medium** Accuracy: **25.13%** Submissions: **441K+** Points: **4**

You are given an integer array **arr**[]. Your task is to find the **smallest positive number** missing from the array.

Note: Positive number starts from 1. The array can have negative integers too.

Examples:

Input: arr[] = [2, -3, 4, 1, 1, 7]
Output: 3
Explanation: Smallest positive missing number is 3.

Input: arr[] = [5, 3, 2, 5, 1]
Output: 4
Explanation: Smallest positive missing number is 4.

Input: arr[] = [-8, 0, -1, -4, -3]
Output: 1
Explanation: Smallest positive missing number is 1.

```java
class Solution {
    public int missingNumber(int[] arr) {
        // code here
        int n=arr.length;
        for(int i=0;i<n;i++)
        {
            while(arr[i]>0&&arr[i]<=n&&arr[arr[i]-1]!=arr[i])
            {
                int temp=arr[i];
                arr[i]=arr[temp-1];
                arr[temp-1]=temp;
            }
        }
        for(int i=0;i<n;i++)
        {
            if(arr[i]!=i+1)
            {
                return i+1;
            }
        }
        return n+1;
    }
}
```