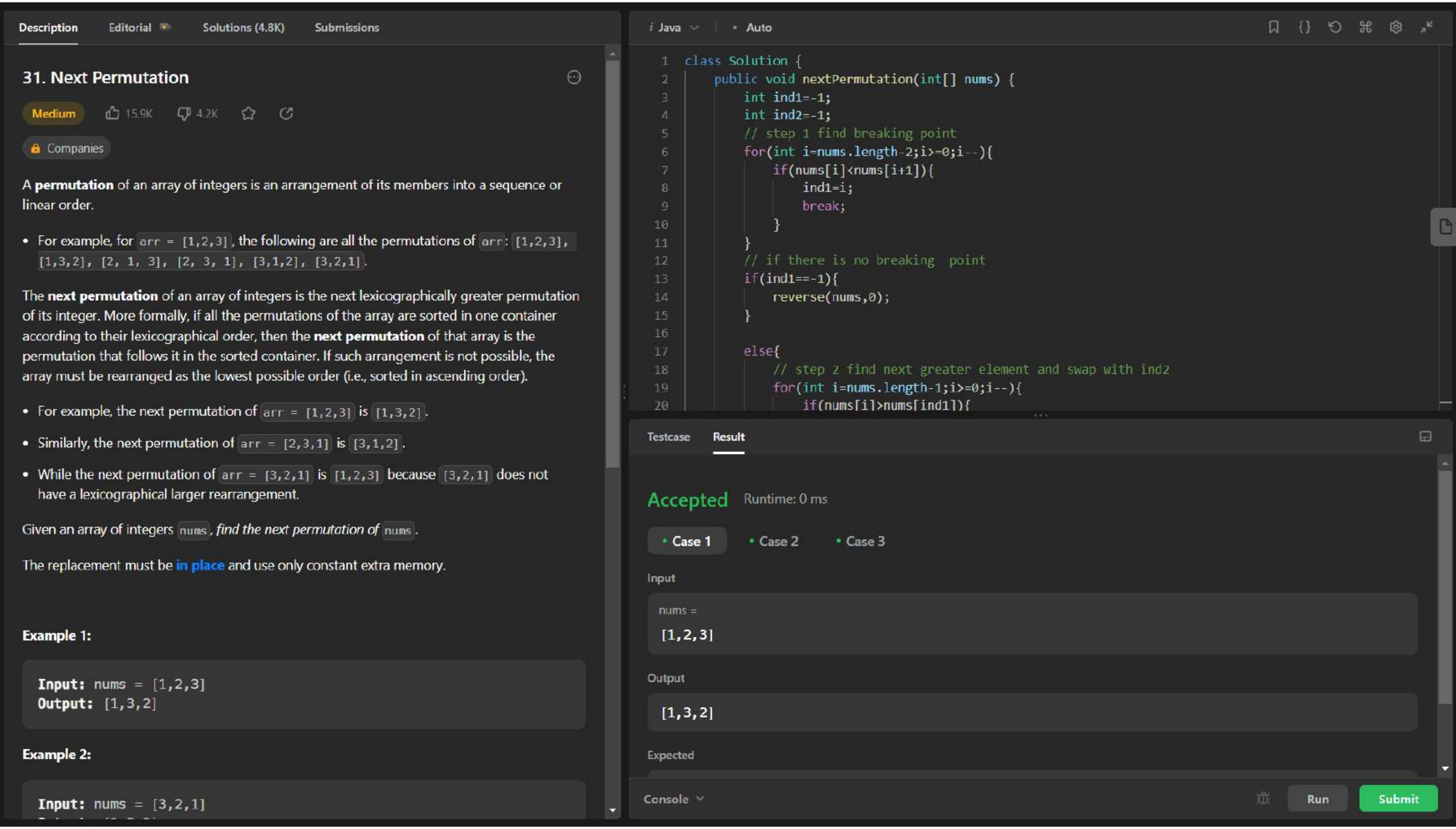


31. Next Permutation

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for `arr = [1,2,3]`, the following are all the permutations of `arr`: `[1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1]`.

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of `arr = [1,2,3]` is `[1,3,2]`.
- Similarly, the next permutation of `arr = [2,3,1]` is `[3,1,2]`.
- While the next permutation of `arr = [3,2,1]` is `[1,2,3]` because `[3,2,1]` does not have a lexicographical larger rearrangement.

Given an array of integers `nums`, *find the next permutation of* `nums`.

The replacement must be **in place** and use only constant extra memory.

Example 1:

```
Input: nums = [1,2,3]
Output: [1,3,2]
```

Example 2:

```
Input: nums = [3,2,1]
```

```java
class Solution {
    public void nextPermutation(int[] nums) {
        int ind1=-1;
        int ind2=-1;
        // step 1 find breaking point
        for(int i=nums.length-2;i>=0;i--){
            if(nums[i]<nums[i+1]){
                ind1=i;
                break;
            }
        }
        // if there is no breaking  point
        if(ind1==-1){
            reverse(nums,0);
        }

        else{
            // step 2 find next greater element and swap with ind2
            for(int i=nums.length-1;i>=0;i--){
                if(nums[i]>nums[ind1]){
```

Accepted Runtime: 0 ms

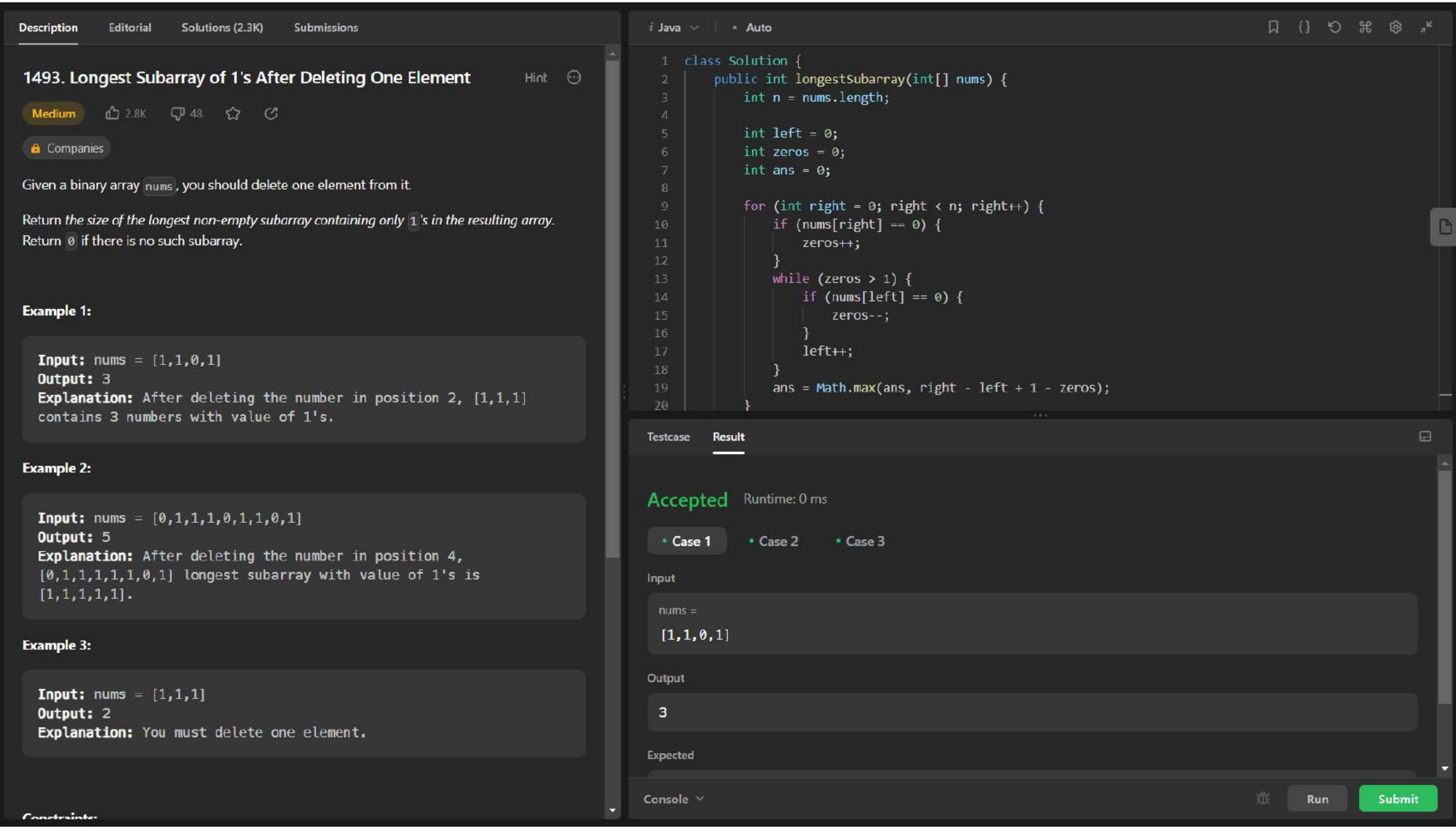
Input

```
nums =
[1,2,3]
```

Output

```
[1,3,2]
```

Expected



1493. Longest Subarray of 1's After Deleting One Element

Given a binary array `nums`, you should delete one element from it.

Return *the size of the longest non-empty subarray containing only* `1`*'s in the resulting array*. Return `0` if there is no such subarray.

Example 1:

```
Input: nums = [1,1,0,1]
Output: 3
Explanation: After deleting the number in position 2, [1,1,1]
contains 3 numbers with value of 1's.
```

Example 2:

```
Input: nums = [0,1,1,1,0,1,1,0,1]
Output: 5
Explanation: After deleting the number in position 4,
[0,1,1,1,1,1,0,1] longest subarray with value of 1's is
[1,1,1,1,1].
```

Example 3:

```
Input: nums = [1,1,1]
Output: 2
Explanation: You must delete one element.
```

Constraints:

```java
class Solution {
    public int longestSubarray(int[] nums) {
        int n = nums.length;

        int left = 0;
        int zeros = 0;
        int ans = 0;

        for (int right = 0; right < n; right++) {
            if (nums[right] == 0) {
                zeros++;
            }
            while (zeros > 1) {
                if (nums[left] == 0) {
                    zeros--;
                }
                left++;
            }
            ans = Math.max(ans, right - left + 1 - zeros);
        }
```

Accepted Runtime: 0 ms

Input

nums =
[1,1,0,1]

Output

3

Expected