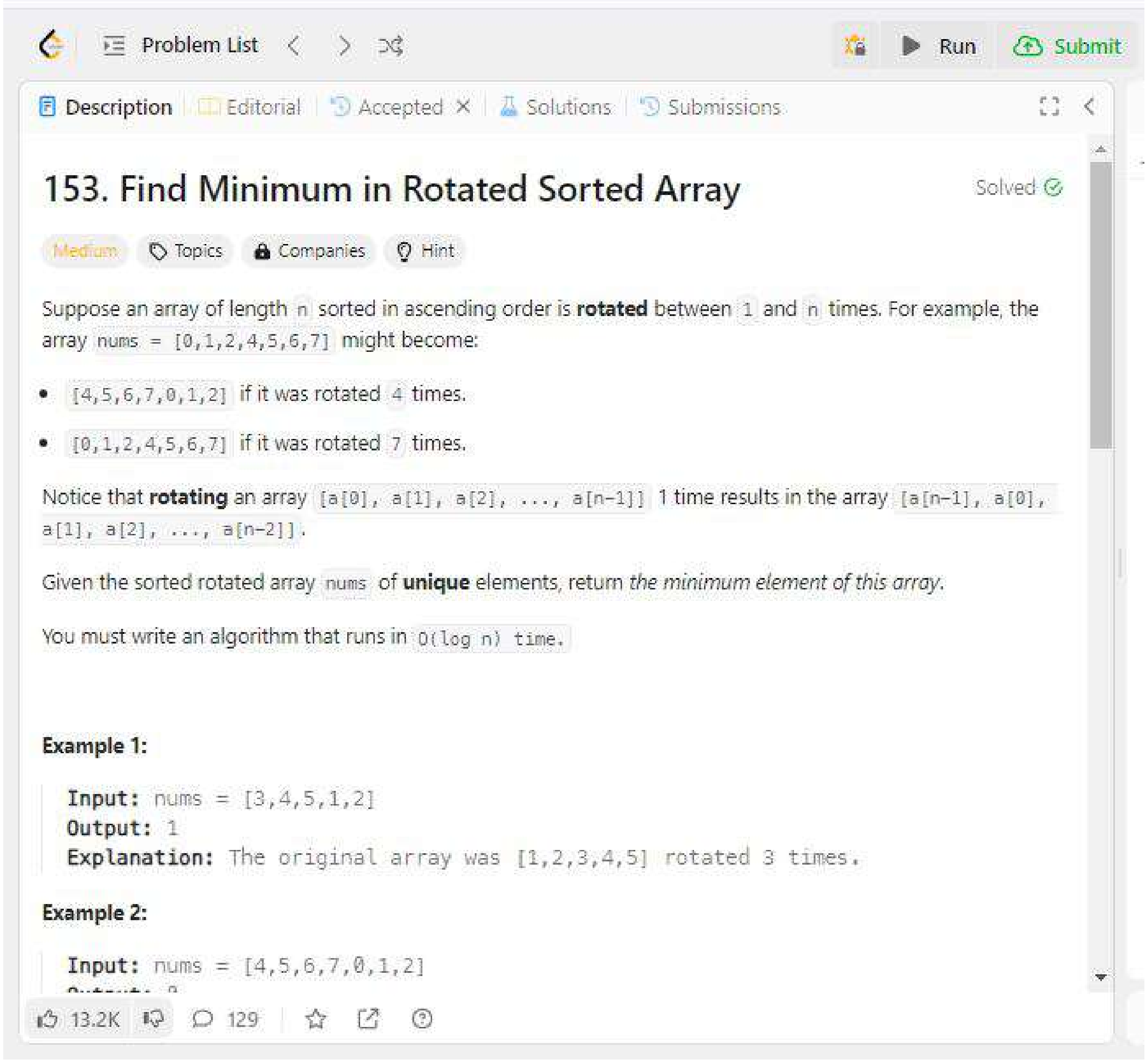


153. Find Minimum in Rotated Sorted Array

Suppose an array of length `n` sorted in ascending order is **rotated** between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated `4` times.
- `[0,1,2,4,5,6,7]` if it was rotated `7` times.

Notice that **rotating** an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in `O(log n) time.`

Example 1:

```
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2]
```

(Find minimum Element in an rotated sorted array)

Examples>

nums= [3,4,5,1,2]

Output: 1

Explanation> The original array was [1,2,3,4,5] rotated 3 times. [1,2,3,4,5] rotation 1 [5,1,2,3,4] rotation 2 [4,5,1,2,3] rotation 3 [3,4,5,1,2]

Approach 1)

We can easily iterate over an array and track min element. But it will take O(N) complexity. We need to solve it in time Complexity O(Log N).

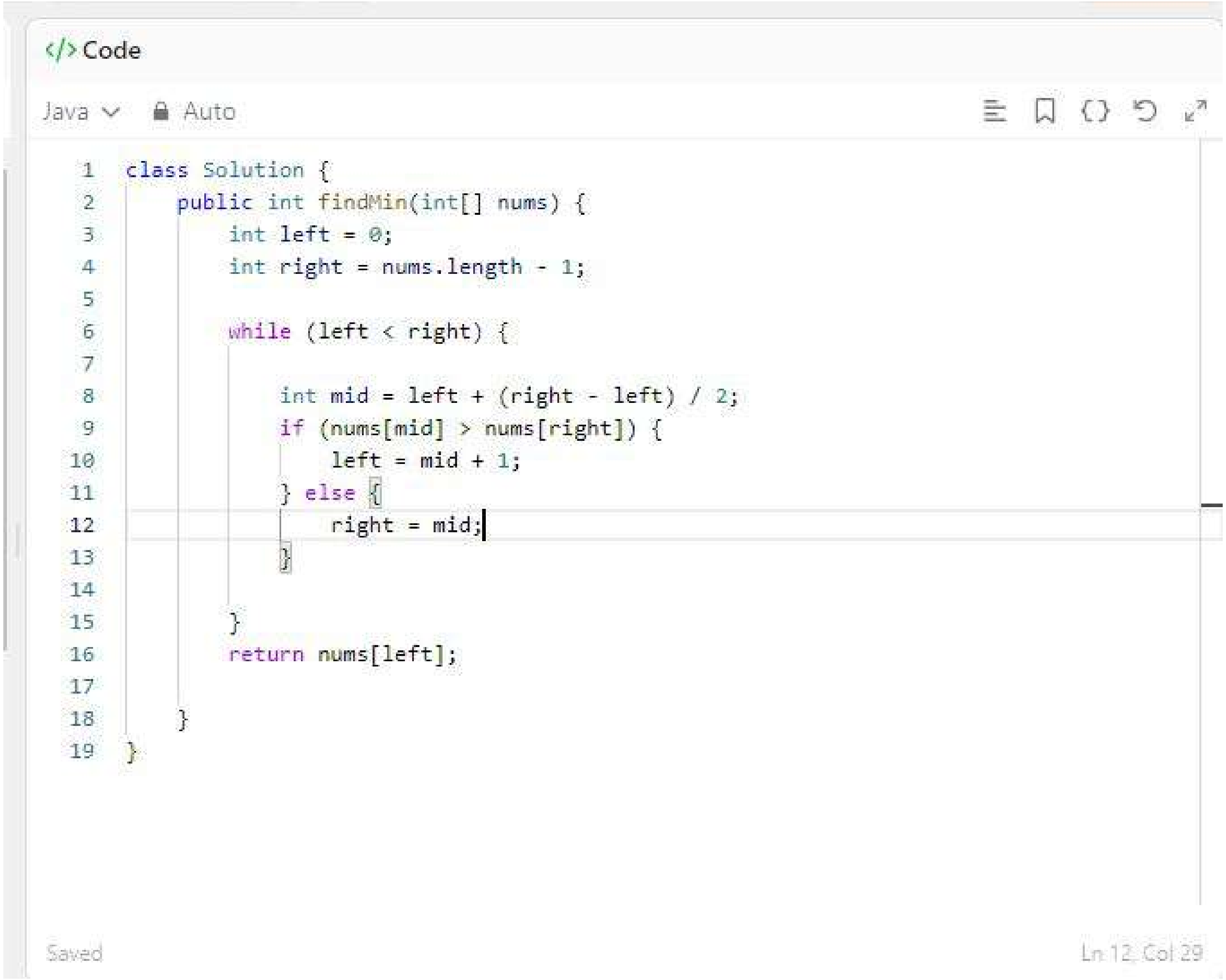
Approach 2)

Because array is sorted array but rotated n times. We will iterate over an array until we get element i+1 less than i. It will be our answer. But in worst case if min elem is at last we will get the answer in O(N) time complexity

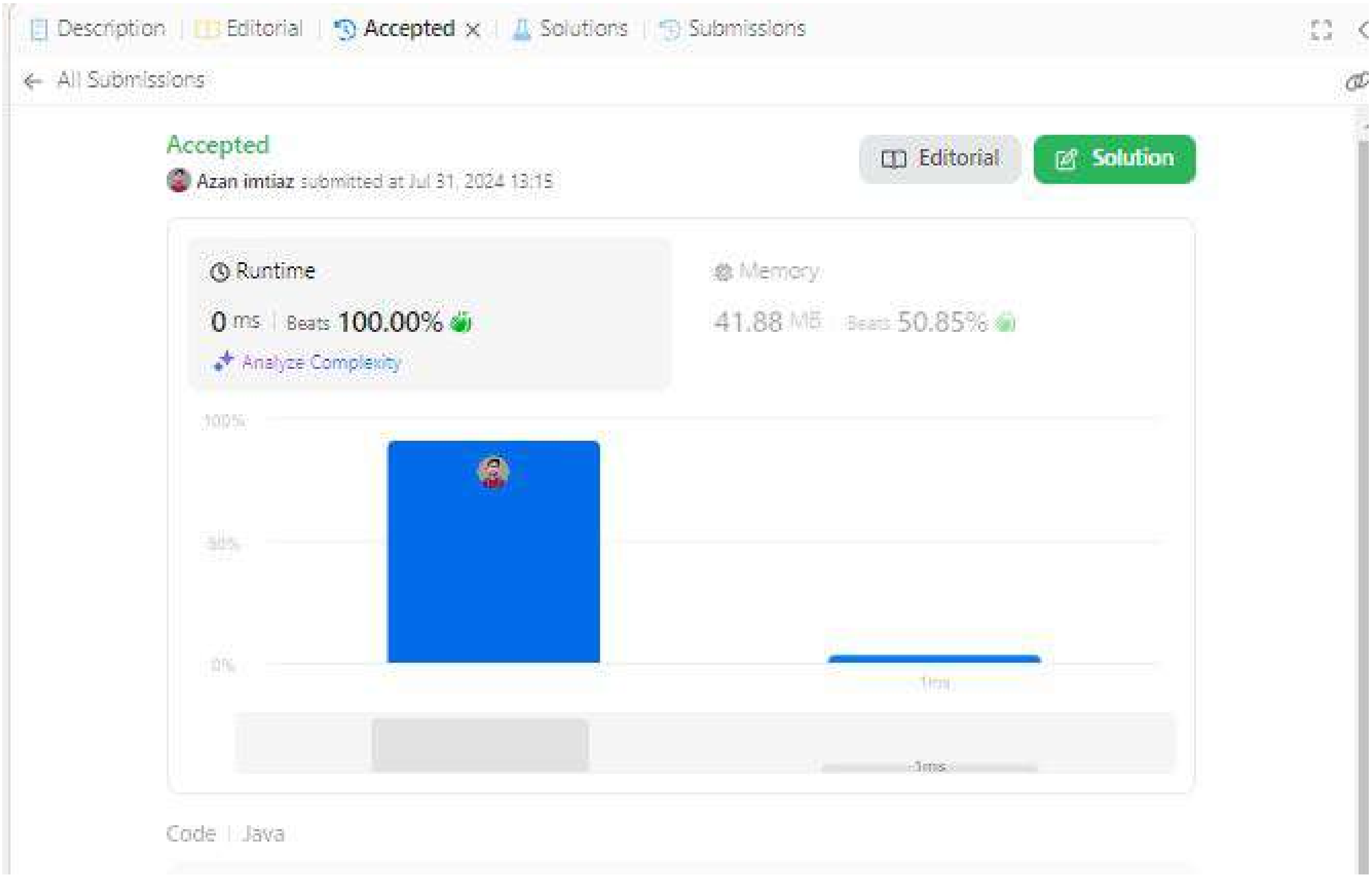
Approach 3)

Apply modified Binary search. find mid if value at mid is ~~less~~ greter than value at right than move left to mid + 1 Else move right to mid. After ~~loo~~ loop end when left is not < right return value at left location

```
public int findMin(int[] nums){
    int left=0;
    int right=nums.length-1;

    while(left< right){
        int mid=left +(right-left)/2;
        if(nums[mid] > nums[right]){
            left=mid+1;}

        else{
            right=mid;
        }
    }
    return nums[left];
  }
}
```



```java
class Solution {
    public int findMin(int[] nums) {
        int left = 0;
        int right = nums.length - 1;

        while (left < right) {

            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right]) {
                left = mid + 1;
            } else {
                right = mid;
            }

        }
        return nums[left];

    }
}
```


Description | Editorial | Accepted × | Solutions | Submissions

← All Submissions

Accepted

Azan imtiaz submitted at Jul 31, 2024 13:15

Editorial | Solution

Runtime

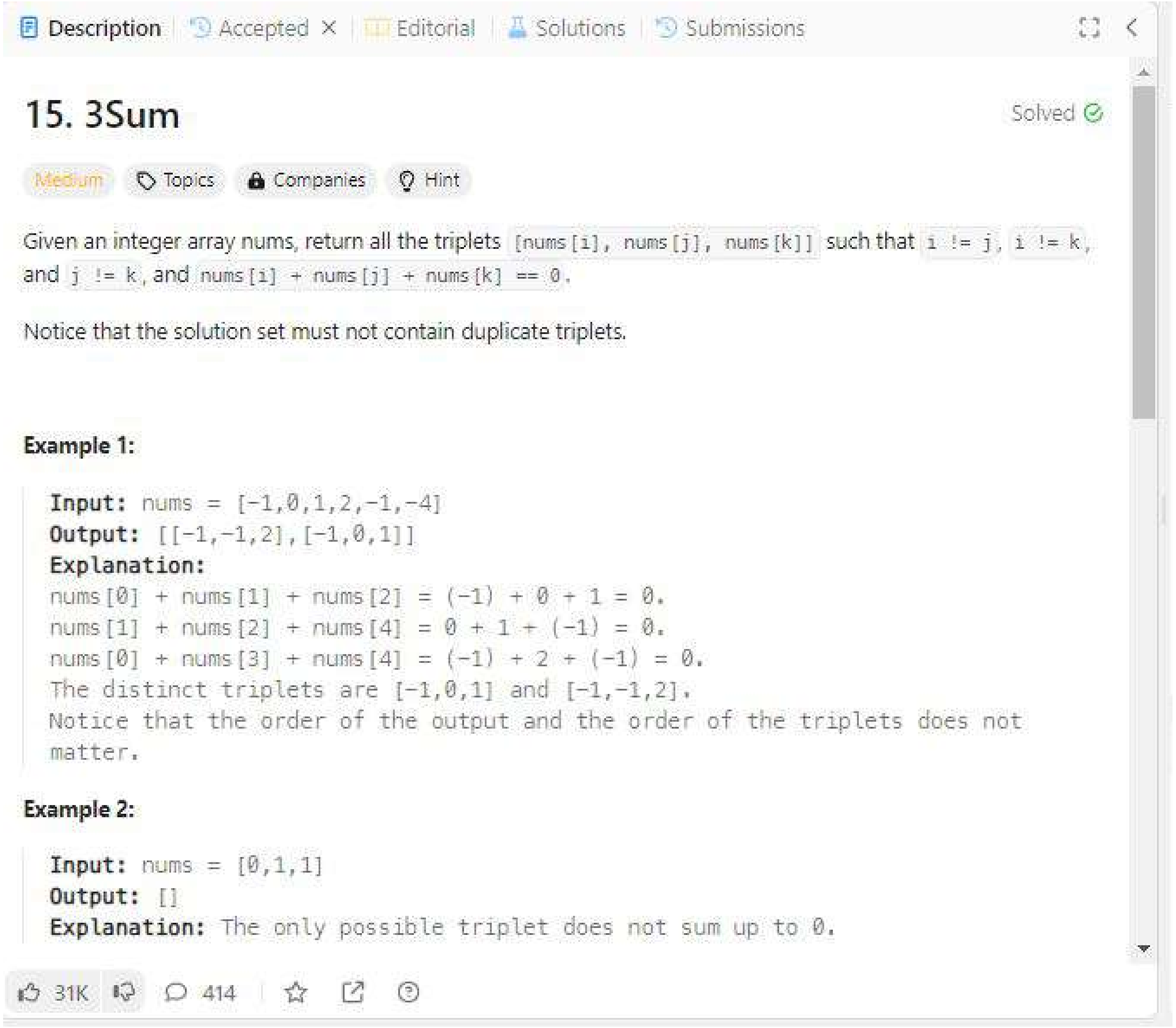
0 ms | Beats 100.00%

Analyze Complexity

Memory

41.88 MB | Beats 50.85%

Code | Java



15. 3Sum

Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not
matter.
```

Example 2:

```
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.
```

Brute Force Approach-

[-1, 0, 1, 2, -1, -4]

Take two elem and find third

[-1, 0, 1, 2, -1, -4]

$O(N^3)$ Time Complexity

Efficient Solution >

first sort an array

~~[illegible]~~. [-4, -1, -1, 0, 1, 2]

fixed		
-4	-1	-1
[-1, -1, 0, 1, 2]	[-1, 0, 1, 2]	[0, 1, 2]
-4 + two value = 0	-1 - 1 + 2 = 0	-1 + 0 + 1 = 0
	-1 + 0 + 1 = 0	

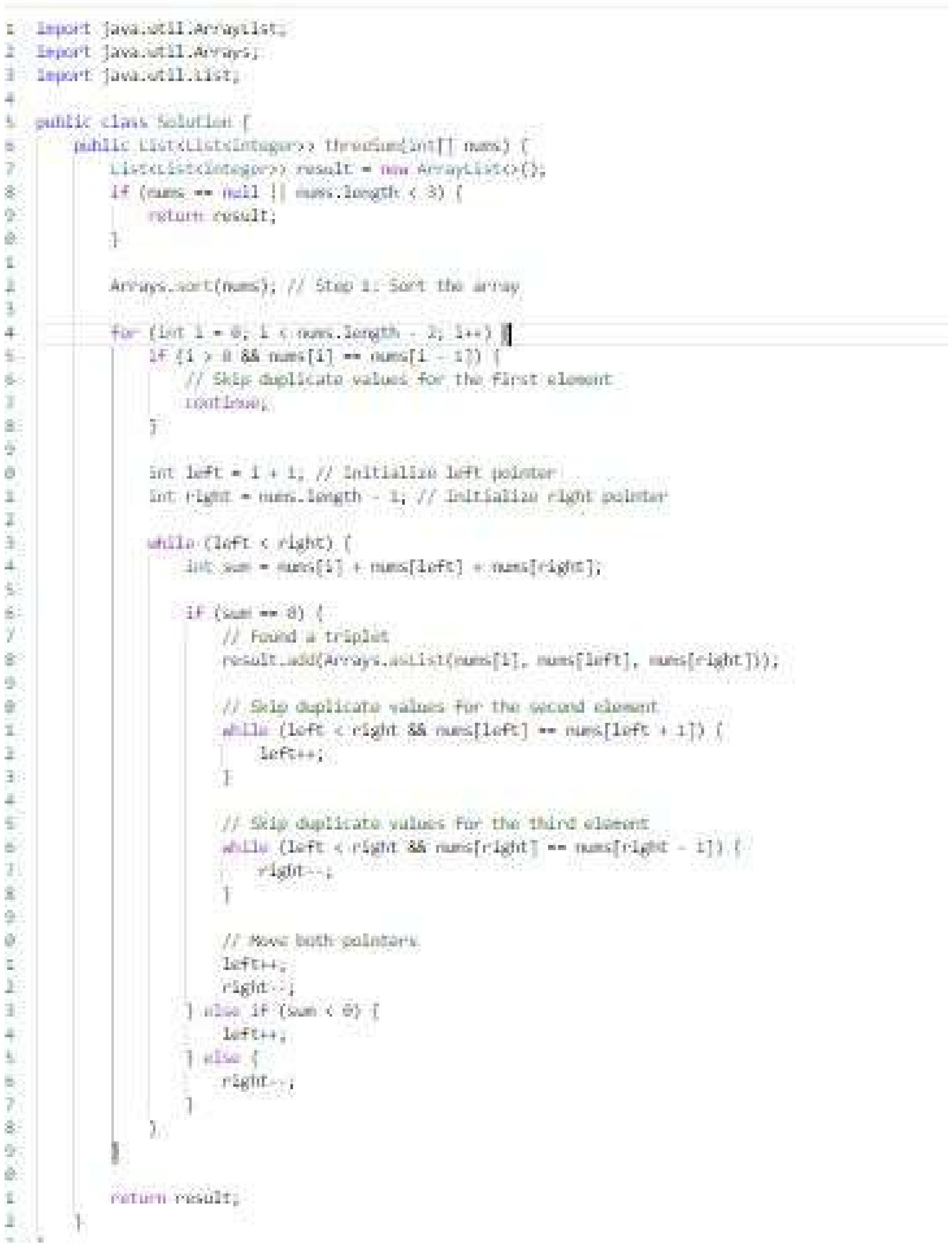
0

[1, 2]

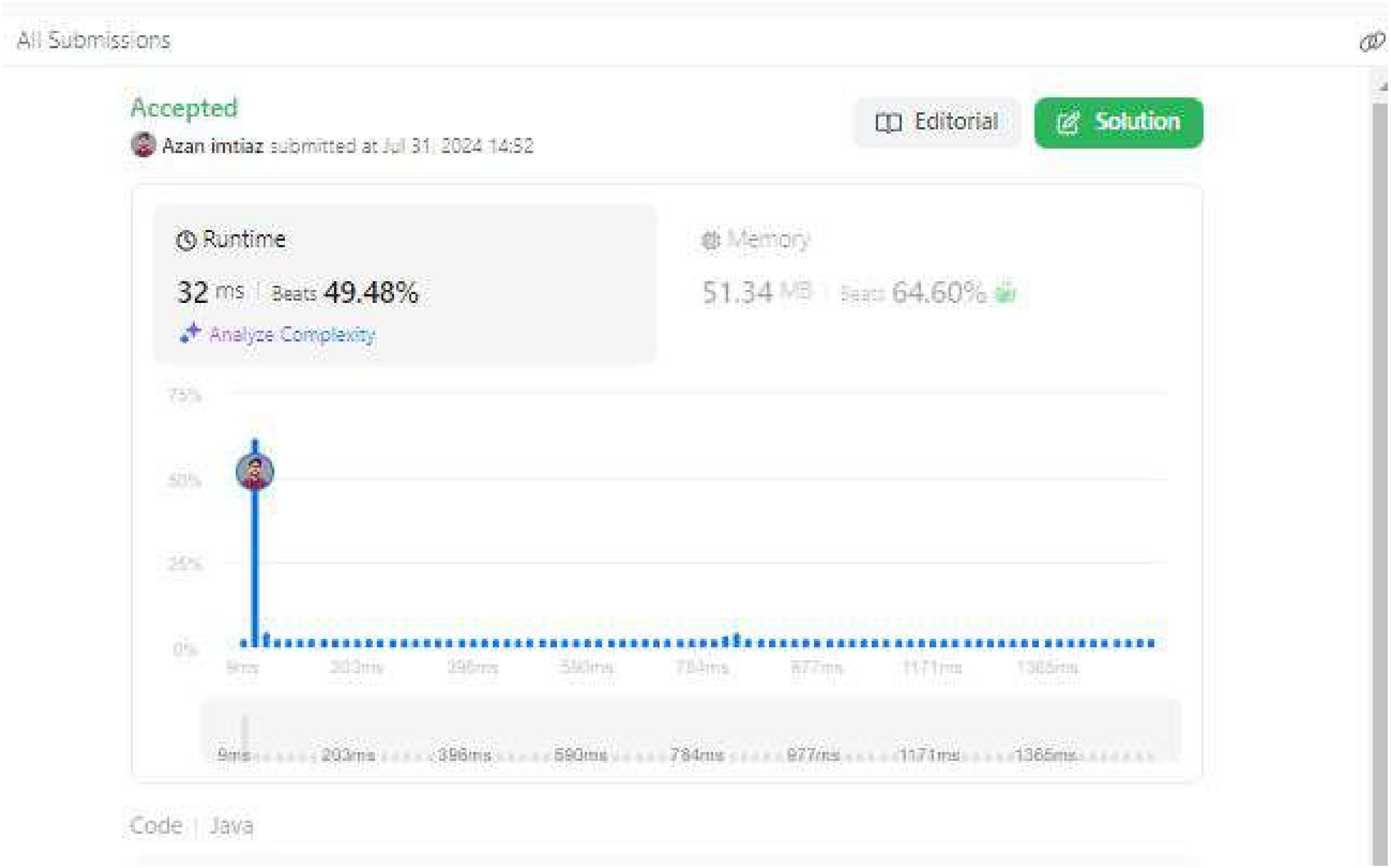
X

so total three triplets

Time Complexity = $O(N^2)$



```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        if (nums == null || nums.length < 3) {
            return result;
        }

        Arrays.sort(nums); // Step 1: Sort the array

        for (int i = 0; i < nums.length - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                // Skip duplicate values for the first element
                continue;
            }

            int left = i + 1; // Initialize left pointer
            int right = nums.length - 1; // Initialize right pointer

            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];

                if (sum == 0) {
                    // Found a triplet
                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));

                    // Skip duplicate values for the second element
                    while (left < right && nums[left] == nums[left + 1]) {
                        left++;
                    }

                    // Skip duplicate values for the third element
                    while (left < right && nums[right] == nums[right - 1]) {
                        right--;
                    }

                    // Move both pointers
                    left++;
                    right--;
                } else if (sum < 0) {
                    left++;
                } else {
                    right--;
                }
            }
        }

        return result;
    }
}
```


All Submissions

Accepted

Azan imtiaz submitted at Jul 31, 2024 14:52

Editorial

Solution

Runtime

32 ms | Beats 49.48%

Analyze Complexity

Memory

51.34 MB | Beats 64.60%

Code | Java