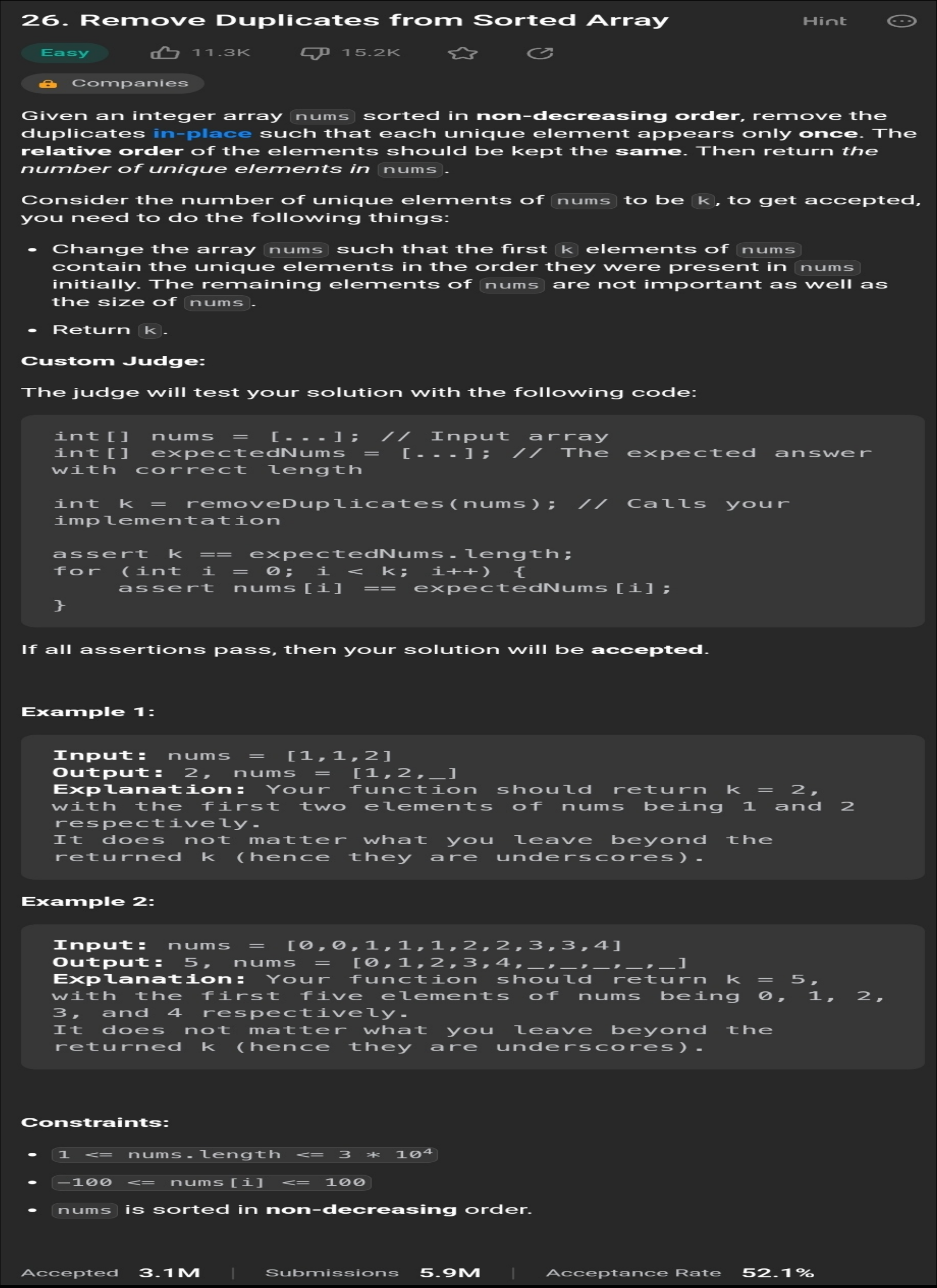


26. Remove Duplicates from Sorted Array

Hint

Easy 11.3K 15.2K

Companies

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates in-place such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* `nums`.

Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer
with correct length

int k = removeDuplicates(nums); // Calls your
implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

Example 1:

```
Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2,
with the first two elements of nums being 1 and 2
respectively.
It does not matter what you leave beyond the
returned k (hence they are underscores).
```

Example 2:

```
Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5,
with the first five elements of nums being 0, 1, 2,
3, and 4 respectively.
It does not matter what you leave beyond the
returned k (hence they are underscores).
```

Constraints:

- `1 <= nums.length <= 3 * 10`^{4}
- `-100 <= nums[i] <= 100`
- `nums` is sorted in **non-decreasing** order.

Accepted **3.1M** | Submissions **5.9M** | Acceptance Rate **52.1%**

Naive approach:

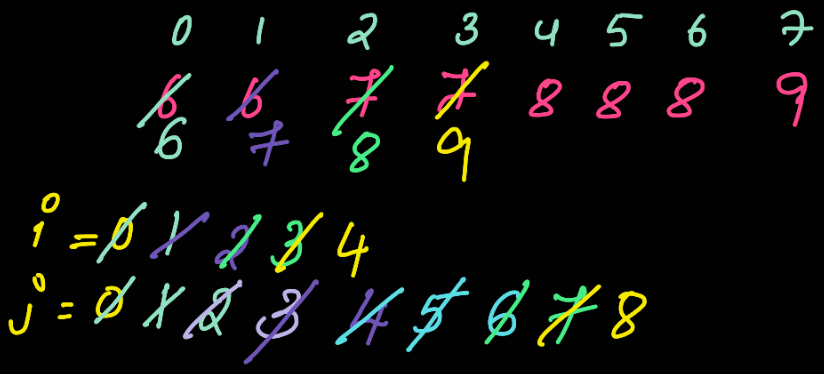
There are different naive ways for solving this problem. We can use hashmap or sets. or we can use an xtra array to store unique elements.

Optimal approach:

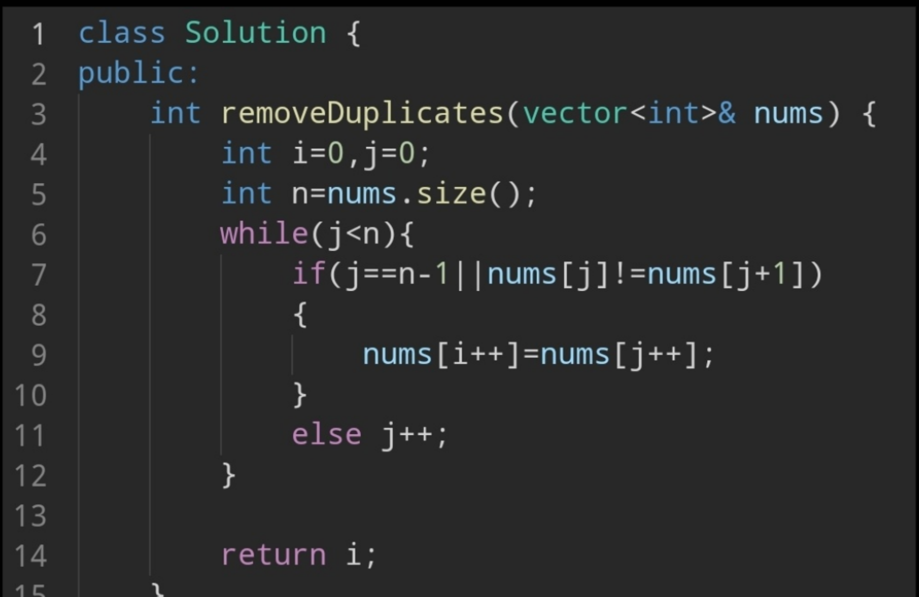
The optimal approach would be using Two pointers. we can use two pointers i & j.

i : That points to the position where the next unique element needs to be placed.

j : This pointer is used to check for unique element in the array using consecutive element comparisions.



0 1 2 3 4 5 6 7
~~6~~ ~~6~~ ~~7~~ ~~7~~ 8 8 8 9
6 7 8 9
i = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4
j = ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ 8

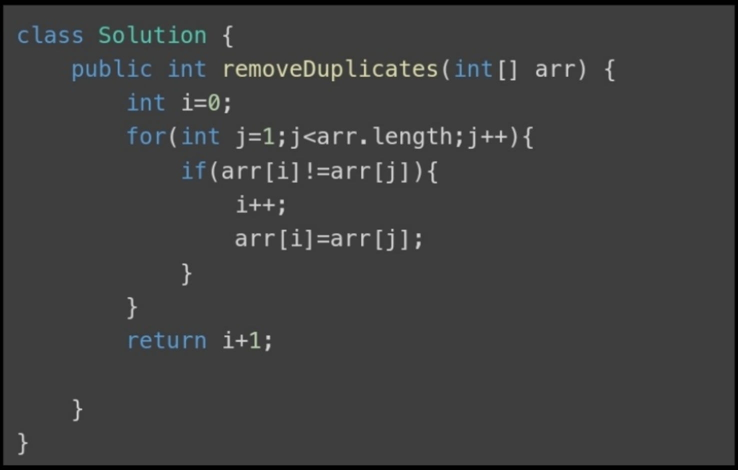


```cpp
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int i=0,j=0;
        int n=nums.size();
        while(j<n){
            if(j==n-1||nums[j]!=nums[j+1])
            {
                nums[i++]=nums[j++];
            }
            else j++;
        }

        return i;
    }
```

```
        }
};
```

Compared to the above code the below code is little more efficient and simple.



```
class Solution {
    public int removeDuplicates(int[] arr) {
        int i=0;
        for(int j=1;j<arr.length;j++){
            if(arr[i]!=arr[j]){
                i++;
                arr[i]=arr[j];
            }
        }
        return i+1;

    }
}
```