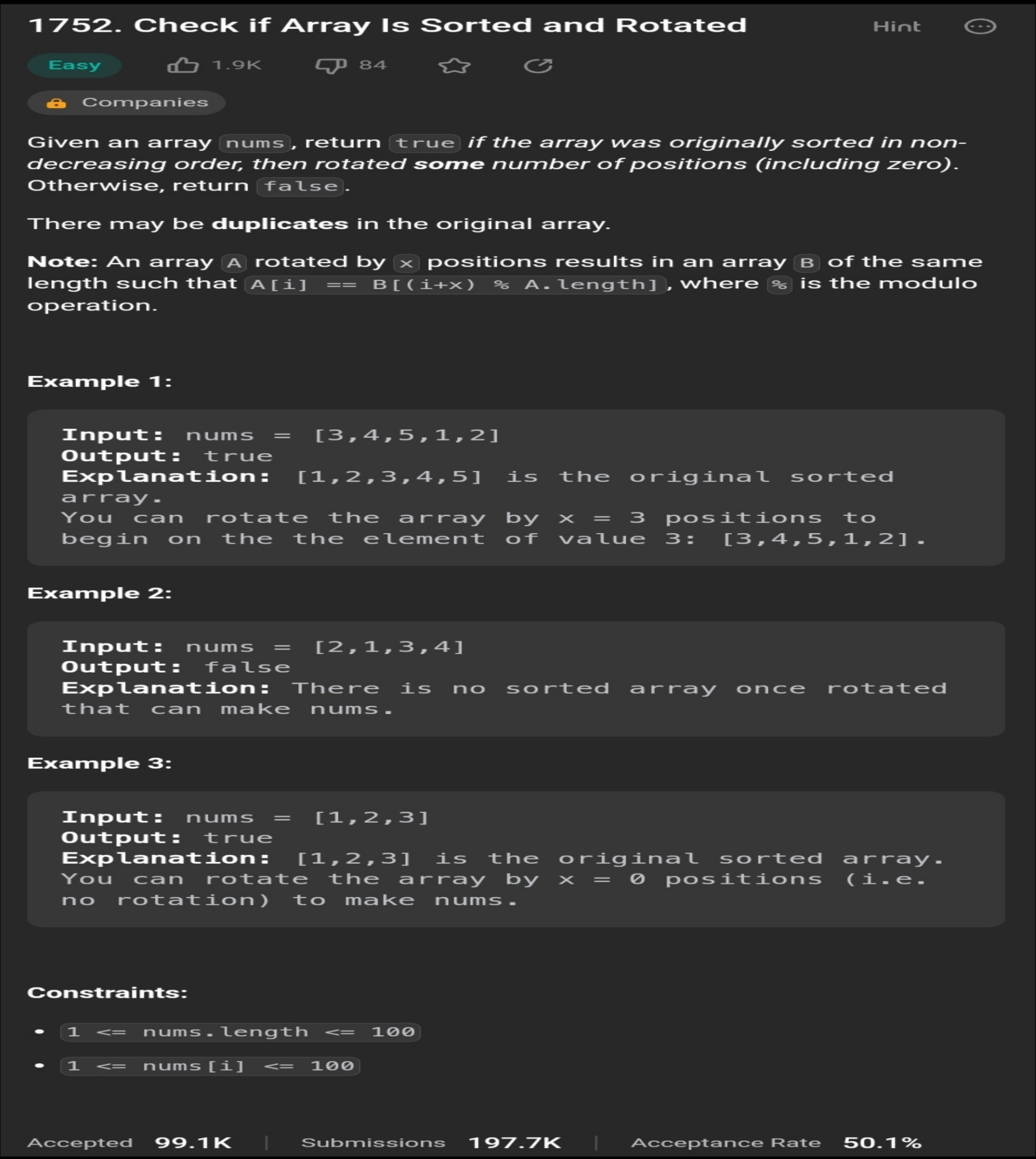


1752. Check if Array Is Sorted and Rotated

Given an array `nums`, return `true` *if the array was originally sorted in non-decreasing order, then rotated* ***some*** *number of positions (including zero)*. Otherwise, return `false`.

There may be **duplicates** in the original array.

Note: An array `A` rotated by `x` positions results in an array `B` of the same length such that `A[i] == B[(i+x) % A.length]`, where `%` is the modulo operation.

Example 1:

```
Input: nums = [3,4,5,1,2]
Output: true
Explanation: [1,2,3,4,5] is the original sorted
array.
You can rotate the array by x = 3 positions to
begin on the the element of value 3: [3,4,5,1,2].
```

Example 2:

```
Input: nums = [2,1,3,4]
Output: false
Explanation: There is no sorted array once rotated
that can make nums.
```

Example 3:

```
Input: nums = [1,2,3]
Output: true
Explanation: [1,2,3] is the original sorted array.
You can rotate the array by x = 0 positions (i.e.
no rotation) to make nums.
```

Constraints:

- `1 <= nums.length <= 100`
- `1 <= nums[i] <= 100`

Accepted 99.1K | Submissions 197.7K | Acceptance Rate 50.1%

Brute force:

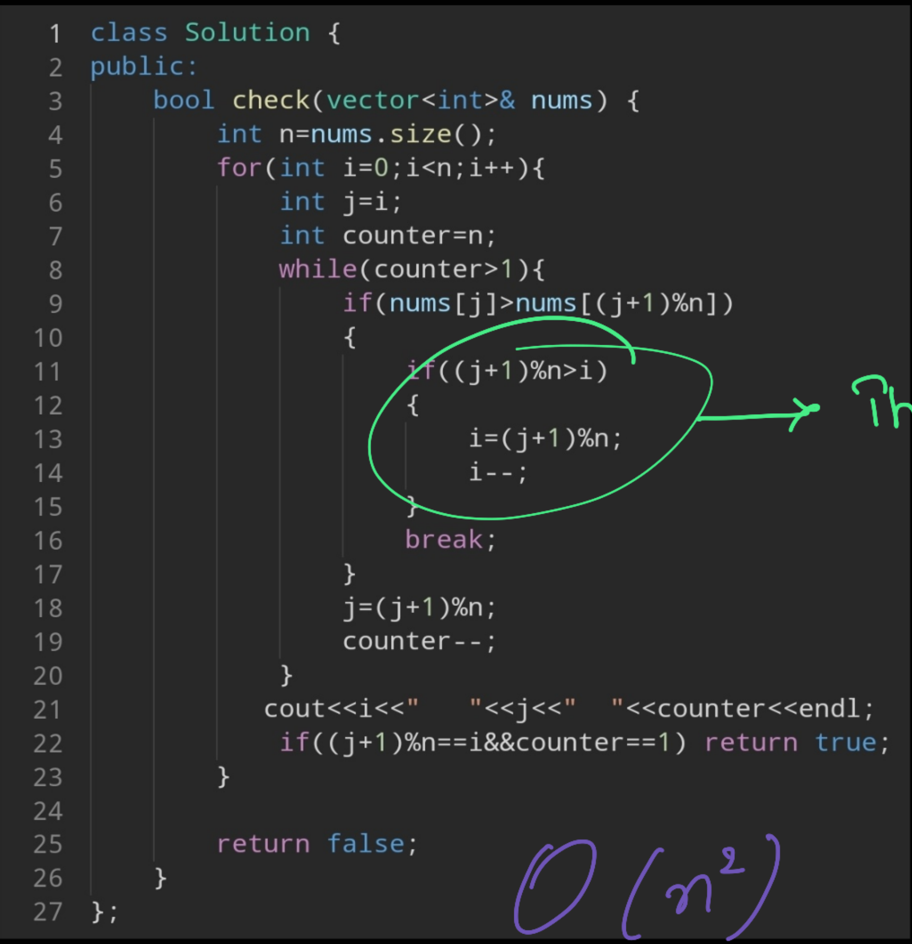
The brute force approach would be to check if there exists sorted order starting from any index.

Eg: 3 4 5 1 2

Sorted order exists from here

sorted order exists from here

if there exists any sorted order from any index then we return true else false.



```cpp
class Solution {
public:
    bool check(vector<int>& nums) {
        int n=nums.size();
        for(int i=0;i<n;i++){
            int j=i;
            int counter=n;
            while(counter>1){
                if(nums[j]>nums[(j+1)%n])
                {
                    if((j+1)%n>i)
                    {
                        i=(j+1)%n;
                        i--;
                    }
                    break;
                }
                j=(j+1)%n;
                counter--;
            }
           cout<<i<<"   "<<j<<"  "<<counter<<endl;
            if((j+1)%n==i&&counter==1) return true;
        }

        return false;
    }
};
```

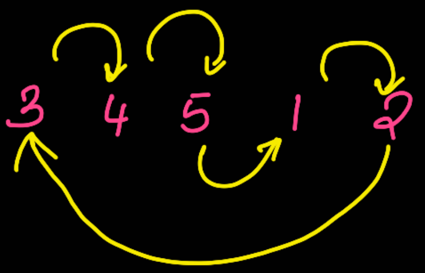
$O(n^2)$

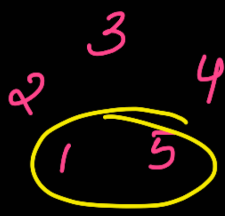
This will add little optimization.

3 4 5 1 2

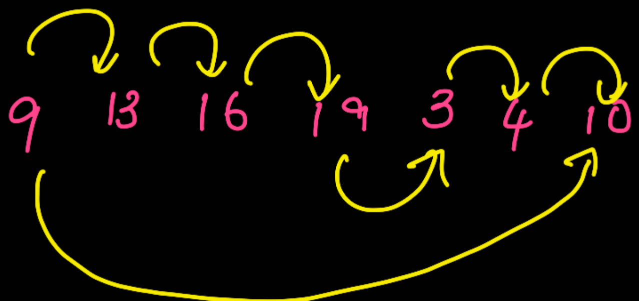
when checking from 3 we break at 5 > 1 which means if we start from 4, then also we break at 5 > 1 so instead of again checking from 4 which anyways is going to break, we skip checking from 4 & 5.

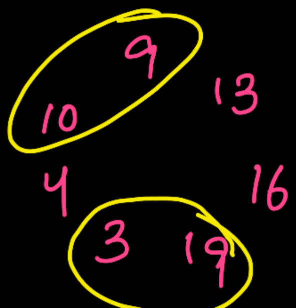
Optimum approach:



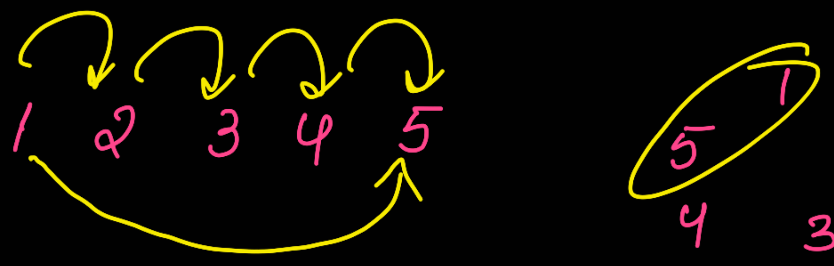


only 1 dip

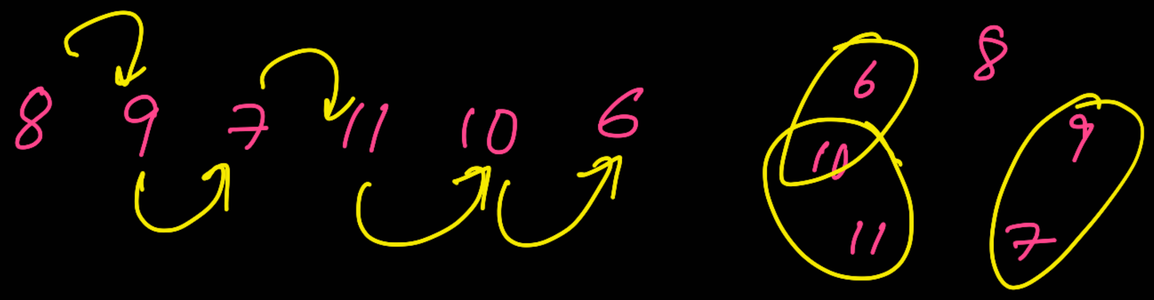




2 dips

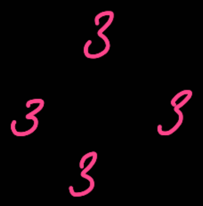


1 dip



3 dips

$3 \to 3 \to 3 \to 3$



0 dips

So if dips ≤ 1 then we can true otherwise we can return false.

```
int dips = 0
for( i : 0 to n-1)
{
      if (a[i] > a[(i+1)%n] )
                  dips ++
}

  return dips ≤ 1
```

O(n)