T.C. o(logn)
S.C. o(1)

Leetcode
Daily Challenge

28/03/2022

problem
Search in Rotated Sorted Array II

problem liked by
3523 people

difficulty
Medium

est. time
5-8 min

Let's build **Intuition**

can be asked in...

34%
Accuracy

Statement

Description

- There is an integer array nums sorted in non-decreasing order (not necessarily with distinct values).
- Before being passed to your function, nums is rotated at an unknown pivot index k (0 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed).

- For example, [0,1,2,4,4,4,5,6,6,7] might be rotated at pivot index 5 and become [4,5,6,6,7,0,1,2,4,4].

- Given the array nums after the rotation and an integer target, return true if target is in nums, or false if it is not in nums.

i/p	o/p
nums = [2,5,6,0,0,1,2], target = 0	true

Observation

What are we given ?

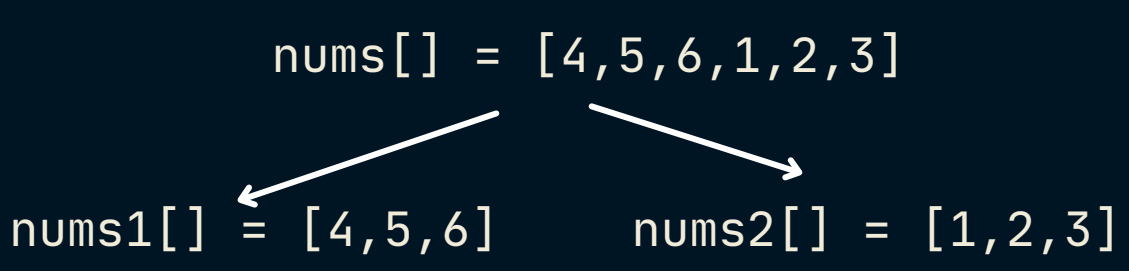
- An array nums[] which was initially sorted but later we rotated it at some index & an integer target

nums[] = [1,2,3,4,5,6] before rotation

nums[] = [4,5,6,1,2,3] after rotation from idx = 3

- You are given the rotated array & return true if target exists

- The rotated nums[] is not exactly sorted but can you see that it can be divided in 2 arrays which are entirely sorted



- Now the above 2 array nums1 & nums2 are sorted & to perform a search on 'Sorted Space' what we use...

Binary Search

Observation

Approach #1

- You can try to divide the search space in 2 parts from the point of rotation & as those spaces will be sorted hence apply 'Binary Search'

- Let's try to look our test-case & get some observation from it...

```
0 1 2 3 4 5
[4 5 0 1 2 3]
```

- Entire array is not sorted but it is sorted in 2parts & we will make use of it

- In Binary Search we jump to mid index in each iteration & then take a decision to discard 1 half of space & keep on searching in other half

- But how we take decision ?
- Consider this nums[] = [1,2,3,4,5,6] target = 6
- now if you jump to idx = 3, you can see that left to idx = 3 all elements are less than 6, so discard the left array & move on to right

- Can you observe one thing, here we were able to make decision because the entire space where your 'mid' lies is 'sorted'

Observation

```
 0 1 2 3 4 5 6 7 8
[4 5 6 1 2 3 4 4 8]
```

- In above example if at some iteration your 'mid' is idx = 4
- now which part of array is 'sorted' ?

```
           0 1 2 3 4 5 6 7 8
          [4 5 6 1 2 3 4 4 4]

left
part      4 5 6 1        right
                         part    3 4 4 4
```

- Your 'left' part of array is not sorted so you can't decide whether your target lies in left part as the power of decision will only work if your space is sorted
- But as 'right' part is sorted so you can choose to discard the array or continue the search inside that.
- Whatever decision you make will be useful for us
- let's explore both the possibilities & get some more observations...

Observation

if mid = 4 (idx)

```
          0 1 2 3 4 5 6 7 8
         [4 5 6 1 2 2 3 4 4]
left
part     4 5 6 1        right part   2 3 4 4
```

- Consider your 'target' = 3
- Since your 'right' array is sorted but how you checked if it is sorted or not...

```
if(nums[mid] <= nums[h])
```

- now the right part may contain 3 or may not, how to check that...

```
if(nums[mid]<target && nums[h]>=target)
```

this condition tells that your target will be b/w mid to h, so put l to right of mid

```
l = mid+1
```

```
else
```

this tell that your target won't lie in this part of array i.e b/w mid to h, so discard this array

```
h = mid-1
```

Observation

What we did in prev. slide

- we jumped to mid & checked which part of array is sorted... left or right
- In prev. example right was sorted.
- then we checked whether or not our target lies in that sorted region.
- if it lies in that sorted part discard the unsorted part if it doesn't lies we discard the sorted part

- So can you see that in every iteration where we may jump(mid) we can still choose to discard one part of array & move to other, so we can use ... Binary Search

Observation

to check if your 'right' part contains target...

```
if(nums[h] >= nums[mid]) {
 if(nums[mid]<target && nums[h]>=target)
    l = mid+1;
 else
   h = mid-1
}
```

to check if your 'left' part contains target...

```
else {
 if(nums[mid]>target && nums[l]<=target)
    h = mid-1;
 else
   l = mid+1
}
```

And it is guaranteed that wherever you may jump(mid idx) one part of array will always appear sorted

But as the elements can be duplicate, we have to handle one case...

Observation

```
        0 1 2 3 4 5 6 7 8
nums = [1,1,1,1,1,2,1,1,1]

       l = 0, h = 8

       consider your mid = 4
```

in above situation when nums[l] == nums[h] == nums[mid], can you make a choice about which part of array will contain your answer...?

As in 'Binary Search' you only have access about l,h,mid & here it seems that every element is same, so you just can't choose one part & discard other, but what can you do incase your nums[mid] != target, simply

```
l++; h--;
```

As we can't discard one half of array in above condition so we just 'linearly' decremented & incremented (l & h), so from this case we can say in worst case our complexity may be

o(n) not o(logn)

```cpp
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int l = 0;
        int h = nums.size() - 1;

        while(l <= h)
        {
            int mid = l + (h-l) / 2;

            if (nums[mid] == target) {
                return true;
            } if((nums[l] == nums[mid]) && (nums[h] == nums[mid])) {
                l++;
                h--;
            } else if(nums[h] >= nums[mid]){
                if((nums[mid] < target) && (nums[h]>= target)) {
                    l = mid + 1;
                } else {
                   h = mid - 1;
                }
            } else {
                if((nums[l] <= target) && (nums[mid] > target)) {
                    h = mid - 1;
                } else {
                    l = mid + 1;
                }

            }
        }
        return false;
    }
};
```

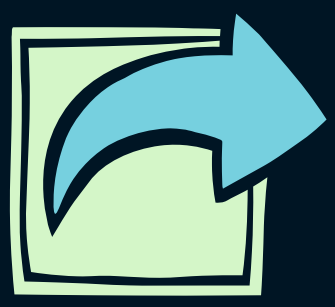
Final thoughts

- Traversing the given matrix requires t.c. of o(m*n) which is unavoidable
- but after that in 1st approach we did sorting or you may use priority_queue they all will take additional(approx) t.c. of o(nlogn) & this is what we avoided & this did step in o(k) in our main solution
- In final solution 2nd loop may look like it has o(m*n) t.c. but if you notice carefully it only runs K times so o(k)

o(m*n) + o(nlogn) => o(m*n) + o(k)

Leave a Like

Comment if you love posts like this, will motivate me to make posts like these



Share, with friends