

81. Search in Rotated Sorted Array II

Medium

Topics

Companies

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` ($0 \leq k < \text{nums.length}$) 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`.

You must decrease the overall operation steps as much as possible.

Example 1:

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

Output: `true`

Example 2:

Input: `nums = [2,5,6,0,0,1,2]`, `target = 3`

Output: `false`

Constraints:

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` is guaranteed to be rotated at some pivot.
- $-10^4 \leq \text{target} \leq 10^4$

Follow up: This problem is similar to [Search in Rotated Sorted Array](#), but `nums` may contain **duplicates**.

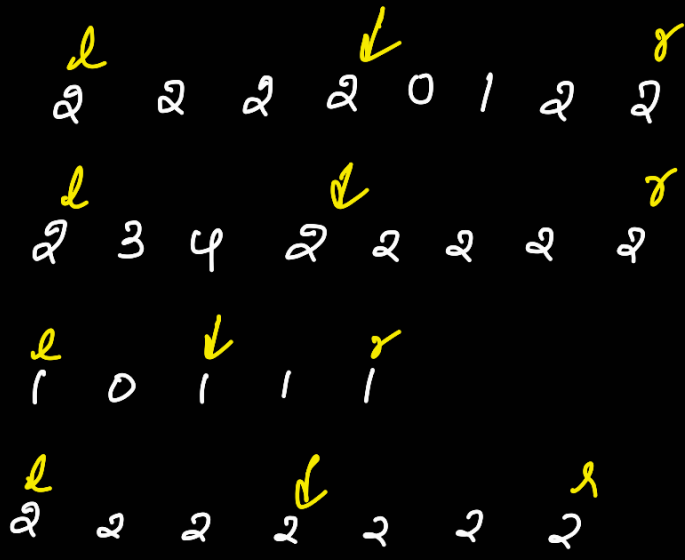
Would this affect the runtime complexity? How and why?

Same thought process as in the case when no duplicates are allowed i.e. #33

But that solution exactly won't work here because duplicates are allowed here.

because duplicates are allowed here.

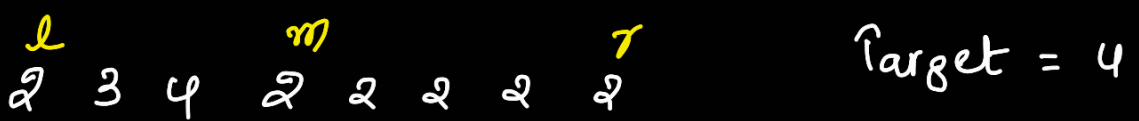
In #33, we were able to decide which side portion of m is in sorted order and which is not based on a_l and a_m . But here in some cases we cannot surely say that so it's not possible to eliminate a side of m completely.



So whenever we find that $a_l == a_m$ it can be any one of the above cases. we cannot identify that and proceed.

So what we do is

instead of eliminating a portion completely, we just reduce the array size by 1 and proceed with binary search.



$a_m \neq \text{Target}$

and $a_l == a_m$ means a_l is also not equal to target.

So just move l pointer by 1 and continue.

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

        while(l<=r){
            int m=(l+r)/2;
            if(nums[m]==target) return true;
            if(nums[l]<nums[m]){ left portion is sorted
                if(target<nums[m] && target>=nums[l]) r=m-1;
                else l=m+1;
            }
            else if(nums[l]==nums[m]){ we don't know
                l++;
            }
            else{ right portion is sorted.
                if(target>nums[m] && target<=nums[r]) l=m+1;
                else r=m-1;
            }
        }

        return false;
    }
};
```

$T(n)$: In most of the cases the Time complexity is $O(\log n)$ but in some cases it can be almost $O(n)$
i.e. 2 2 2 2 2 2 2 2
(or)

2 2 2 2 1 2 2

So

Bestcase $T(n)$: $O(\log n)$

Avgcase $T(n)$: $O(\log n)$

Worstcase $T(n)$: $O(n)$