Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array":
What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

Solution 1

Since we will have some duplicate elements in this problem, it is a little tricky because sometimes we cannot decide whether to go to the left side or right side. So for this condition, I have to probe both left and right side simultaneously to decide which side we need to find the number. Only in this condition, the time complexity may be *O(n)*. The rest conditions are always *O(log n)*.

For example:

input: `113111111111`, Looking for *target* `3`.

Is my solution correct? My code is as followed:

```java
public class Solution {
    public boolean search(int[] A, int target) {
        // IMPORTANT: Please reset any member data you declared, as
        // the same Solution instance will be reused for each test case.
        int i = 0;
        int j = A.length - 1;
        while(i <= j){
            int mid = (i + j) / 2;
            if(A[mid] == target)
                return true;
            else if(A[mid] < A[i]){
                if(target > A[j])
                    j = mid - 1;
                else if(target < A[mid])
                    j = mid - 1;
                else
                    i = mid + 1;
            }else if(A[mid] > A[i]){
                if(target < A[mid] && target >= A[i])
                    j = mid - 1;
                else
                    i = mid + 1;
            }else{ // A[mid] == A[i]
                if(A[mid] != A[j])
                    i = mid + 1;
                else{
                    boolean flag = true;
                    for(int k = 1; mid - k >= i && mid + k <= j; k++){
                        if(A[mid] != A[mid - k]){
                            j = mid - k;
                            flag = false;
                            break;
                        }else if(A[mid] != A[mid + k]){
                            i = mid + k;
                            flag = false;
                            break;
                        }
                    }
                    if(flag)
                        return false;
                }
            }
        }
        return false;
    }
}
```

written by baojialiang original link here

Solution 2

```
class Solution {
public:
  bool search(int A[], int n, int target) {
    int lo =0, hi = n-1;
    int mid = 0;
    while(lo<hi){
          mid=(lo+hi)/2;
          if(A[mid]==target) return true;
          if(A[mid]>A[hi]){
              if(A[mid]>target && A[lo] <= target) hi = mid;
              else lo = mid + 1;
          }else if(A[mid] < A[hi]){
              if(A[mid]<target && A[hi] >= target) lo = mid + 1;
              else hi = mid;
          }else{
              hi--;
          }

    }
    return A[lo] == target ? true : false;
  }
};
```

written by ggyc1993 original link here

Solution 3

The idea is the same as the previous one without duplicates

```
1) everytime check if targe == nums[mid], if so, we find it.
2) otherwise, we check if the first half is in order (i.e. nums[left]<=nums[mid])

  and if so, go to step 3), otherwise, the second half is in order,   go to step
4)
3) check if target in the range of [left, mid-1] (i.e. nums[left]<=target < nums[
mid]), if so, do search in the first half, i.e. right = mid-1; otherwise, search
in the second half left = mid+1;
4)  check if target in the range of [mid+1, right] (i.e. nums[mid]<target <= nums
[right]), if so, do search in the second half, i.e. left = mid+1; otherwise searc
h in the first half right = mid-1;
```

The only difference is that due to the existence of duplicates, we can have nums[left] == nums[mid] and in that case, the first half could be out of order (i.e. NOT in the ascending order, e.g. [3 1 2 3 3 3 3]) and we have to deal this case separately. In that case, it is guaranteed that nums[right] also equals to nums[mid], so what we can do is to check if nums[mid]== nums[left] == nums[right] before the original logic, and if so, we can move left and right both towards the middle by 1. and repeat.

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

        while(left<=right)
        {
            mid = (left + right) >> 1;
            if(nums[mid] == target) return true;

            // the only difference from the first one, trickly case, just updat l
eft and right
            if( (nums[left] == nums[mid]) && (nums[right] == nums[mid]) ) {++left
; --right;}

            else if(nums[left] <= nums[mid])
            {
                if( (nums[left]<=target) && (nums[mid] > target) ) right = mid-1;
                else left = mid + 1;
            }
            else
            {
                if((nums[mid] < target) &&  (nums[right] >= target) ) left = mid+
1;
                else right = mid-1;
            }
        }
        return false;
    }
};
```

written by dong.wang.1694 original link here