Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\10.Search_in_rotated_sorted_array_with_duplicates.cpp

```
/*
QUESTION:
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.

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
*/

/*
APPROACH:
We can modify the standard binary search algorithm to search for the target element.
1. Initialize low = 0 and high = nums.size() - 1.
2. While low <= high, calculate mid = low + (high - low) / 2.
3. If nums[mid] equals the target, return true.
4. If nums[mid] is equal to nums[low], we are in a situation where we can't determine which part of the array is sorted.
   In this case, we increment low and decrement high to skip the duplicate elements.
5. If the left part of the array from low to mid is sorted, check if the target lies within this range.
   If so, update high = mid - 1. Otherwise, update low = mid + 1.
6. If the right part of the array from mid to high is sorted, check if the target lies within this range.
   If so, update low = mid + 1. Otherwise, update high = mid - 1.
7. If the target is not found, return false.

CODE:
*/

bool search(vector<int>& nums, int target) {
    int low = 0, high = nums.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target)
            return true;
        if (nums[mid] == nums[low] && nums[mid] == nums[high]) {
```

```
            low++;
            high--;
            continue;
        }
        if (nums[low] <= nums[mid]) {
            if (nums[low] <= target && target <= nums[mid])
                high = mid - 1;
            else
                low = mid + 1;
        } else {
            if (nums[mid] <= target && target <= nums[high])
                low = mid + 1;
            else
                high = mid - 1;
        }
    }
    return false;
}

// TIME COMPLEXITY: O(log n)
// SPACE COMPLEXITY: O(1)

```