Search Element in Rotated Sorted Array II

Problem Statement: Given an integer array **arr** of size **N**, sorted in ascending order (**may contain duplicate values**) and a target value **k**. Now the array is rotated at some pivot point unknown to you. Return True if **k** is present and otherwise, return False.

Pre-requisite: Search Element in Rotated Sorted Array I **&** Binary Search algorithm

▼ **Examples**

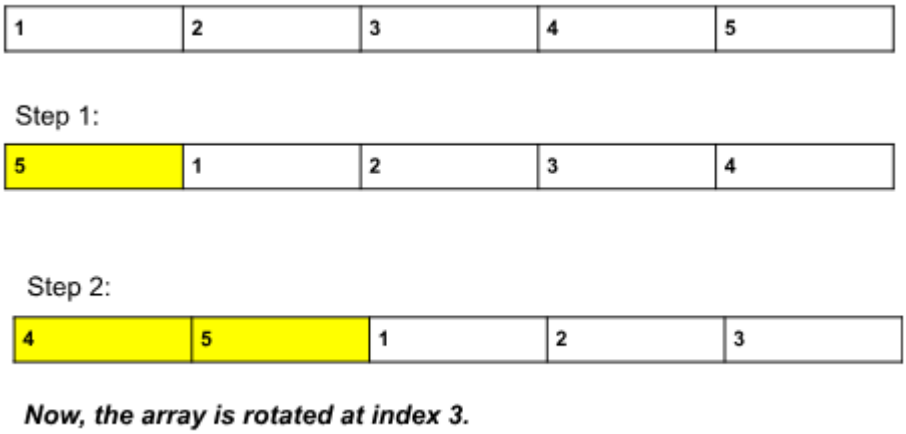
```
Example 1:
Input Format: arr = [7, 8, 1, 2, 3, 3, 3, 4, 5, 6], k = 3
Result: True
Explanation: The element 3 is present in the array. So, the answer is True.

Example 2:
Input Format: arr = [7, 8, 1, 2, 3, 3, 3, 4, 5, 6], k = 10
Result: False
Explanation: The element 10 is not present in the array. So, the answer is False.
```

Solution:

How does the rotation occur in a sorted array?

Let's consider a sorted array: {1, 2, 3, 4, 5}. If we rotate this array at index 3, it will become: {4, 5, 1, 2, 3}. In essence, we moved the element at the last index to the front, while shifting the remaining elements to the right. We performed this process twice.



Now, the array is rotated at index 3.

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

▼ Brute Force Approach
▼ Algorithm / Intuition ›

Naive Approach (Brute force):

One straightforward approach we can consider is using the linear search algorithm. Using this method, we will traverse the array to check if the target is present in the array. If it is found we will simply return True and otherwise, we will return False.

Algorithm:

- We will traverse the array and check every element if it is equal to k. If we find any element, we will return True.
- **Otherwise,** we will return False.

▼ Code ›

```
from typing import *
def searchInARotatedSortedArrayII(arr : List[int], k : int) -> bool:
    for num in arr:
        if num == k:
            return True
    return False

if __name__ == "__main__":
    arr = [7, 8, 1, 2, 3, 3, 3, 4, 5, 6]
    k = 3
    ans = searchInARotatedSortedArrayII(arr, k)
    if not ans:
        print("Target is not present.")
    else:
        print("Target is present in the array.")
```

Output: Target is present in the array.

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.**Reason:** We have to iterate through the entire array to check if the target is present in the array.

Space Complexity: O(1)**Reason:** We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Optimal Approach
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

Like the previous problem, we will use the Binary Search algorithm to solve this problem.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

In the previous problem, in order to efficiently search for the target value, we followed a simple two-step process.

- First, we identify the sorted half of the array.
- Once found, we determine if the target is located within this sorted half.
    - If not, we eliminate that half from further consideration.
    - Conversely, if the target does exist in the sorted half, we eliminate the other half.

Let's observe how we identify the sorted half:

We basically compare arr[mid] with arr[low] and arr[high] in the following way:

- **If arr[low] <= arr[mid]:** In this case, we identified that the left half is sorted.
- **If arr[mid] <= arr[high]:** In this case, we identified that the right half is sorted.

This check was effective in the previous problem, where there were no duplicate numbers. However, in the current problem, the array may contain duplicates. ***Consequently, the previous approach will not work when arr[low] = arr[mid] = arr[high].***

How to handle the edge case arr[low] = arr[mid] = arr[high]:

In the algorithm, we first check if arr[mid] is the target before identifying the sorted half. If arr[mid] is not our target, we encounter this edge case. In this scenario, since arr[mid] = arr[low] = arr[high], it means that neither arr[low] nor arr[high] can be the target. To handle this edge case, we simply remove arr[low] and arr[high] from our search space, without affecting the original algorithm.

To eliminate elements arr[low] and arr[high], we can achieve this by simply incrementing the low pointer and decrementing the high pointer by one step. We will continue this process until the condition arr[low] = arr[mid] = arr[high] is no longer satisfied.

Now, we will remove arr[low] and arr[high] from the search space

Now, the condition, arr[low] = arr[mid] = arr[high] is no longer satisfied.

Note: *As long as this condition is met, we will skip the steps of determining the sorted half and eliminating one of the halves based on the target's location. Instead, we will solely focus on eliminating arr[low] and arr[high].*

We will apply the same algorithm as the previous problem by just adding an extra check to handle the above edge case.

Algorithm:

The steps are as follows:

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers like this: low will point to the first index, and high will point to the last index.
2. **Calculate the 'mid':** Now, inside a loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
3. **Check if arr[mid] = target:** If it is, return True.
4. **Check if arr[low] = arr[mid] = arr[high]:** If this condition is satisfied, we will just increment the low pointer and decrement the high pointer by one step. We will not perform the later steps until this condition is no longer satisfied. So, we will ***continue to the next iteration from this step***.
5. Identify the sorted half, check where the target is located, and then eliminate one half accordingly:
    1. **If arr[low] <= arr[mid]:** This condition ensures that the left part is sorted.
        1. **If arr[low] <= target && target <= arr[mid]:** It signifies that the target is in this sorted half. So, we will eliminate the right half (***high = mid-1***).
        2. **Otherwise,** the target does not exist in the sorted half. So, we will eliminate this left half by doing **low = mid+1**.
    2. **Otherwise, if the right half is sorted:**
        1. **If arr[mid] <= target && target <= arr[high]:** It signifies that the target is in this sorted right half. So, we will eliminate the left half (***low = mid+1***).
        2. **Otherwise,** the target does not exist in this sorted half. So, we will eliminate this right half by doing **high = mid-1**.
6. Once, the 'mid' points to the target, we will return True.
7. This process will be inside a loop and the loop will continue until low crosses high. If no element is found, we will return False.

Dry-run: *Please refer to the video for it.*

▼ Code ›

```
from typing import *
def searchInARotatedSortedArrayII(arr : List[int], k : int) -> bool:
    n = len(arr)  # size of the array
    low, high = 0, n - 1

    while low <= high:
        mid = (low + high) // 2

        # if mid points to the target
        if arr[mid] == k:
            return True

        # Edge case:
        if arr[low] == arr[mid] and arr[mid] == arr[high]:
            low += 1
            high -= 1
            continue

        # if left part is sorted
        if arr[low] <= arr[mid]:
            if arr[low] <= k <= arr[mid]:
                # element exists
                high = mid - 1
            else:
                # element does not exist
                low = mid + 1
        else:  # if right part is sorted
            if arr[mid] <= k <= arr[high]:
                # element exists
                low = mid + 1
            else:
                # element does not exist
                high = mid - 1

    return False

if __name__ == "__main__":
    arr = [7, 8, 1, 2, 3, 3, 3, 4, 5, 6]
    k = 3
    ans = searchInARotatedSortedArrayII(arr, k)
    if not ans:
        print("Target is not present.")
    else:
        print("Target is present in the array.")
```

Output: Target is present in the array.

▼ Complexity Analysis ›

Time Complexity: O(logN) for the best and average case. **O(N/2)** for the worst case. Here, N = size of the given array.

Reason: In the best and average scenarios, the binary search algorithm is primarily utilized and hence the time complexity is O(logN). However, in the worst-case scenario,

where all array elements are the same but not the target (e.g., given array = {3, 3, 3, 3, 3, 3, 3}), we continue to reduce the search space by adjusting the low and high pointers until they intersect. This worst-case situation incurs a time complexity of O(N/2).

Space Complexity: O(1)**Reason:** We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).