Search Element in a Rotated Sorted Array

Problem Statement: Given an integer array arr of size N, sorted in ascending order (with distinct values) and a target value k. Now the array is rotated at some pivot point unknown to you. Find the index at which k is present and if k is not present return -1.

▼ **Examples**

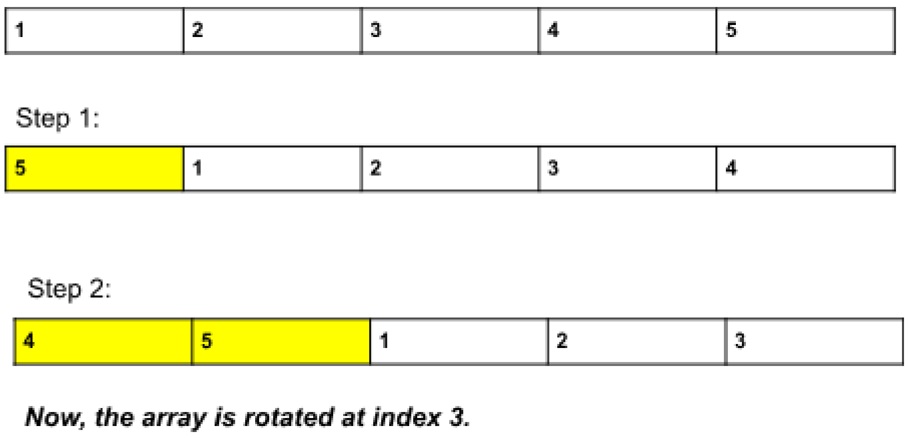
```
Example 1:
Input Format: arr = [4,5,6,7,0,1,2,3], k = 0
Result: 4
Explanation: Here, the target is 0. We can see that 0 is present in the given rotated sorted array,
nums. Thus, we get output as 4, which is the index at which 0 is present in the array.

Example 2:
Input Format: arr = [4,5,6,7,0,1,2], k = 3
Result: -1
Explanation: Here, the target is 3. Since 3 is not present in the given rotated sorted array. Thus, we
get the output as -1.
```

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.



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

Practice: [Solve Problem]

▼ 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 find the location of the target value. If it is found we will simply return the index and otherwise, we will return -1.

Algorithm:

- We will traverse the array and check every element if it is equal to k. If we find any element, we will return its index.
- **Otherwise,** we will return -1.

▼ Code ›

```
def search(arr, n, k):
    for i in range(n):
        if arr[i] == k:
            return i
    return -1

if __name__ == "__main__":
    arr = [7, 8, 9, 1, 2, 3, 4, 5, 6]
    n = 9
    k = 1
    ans = search(arr, n, k)
    if ans == -1:
        print("Target is not present.")
    else:
        print("The index is:", ans)
```

Output: The index is: 3

▼ 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):

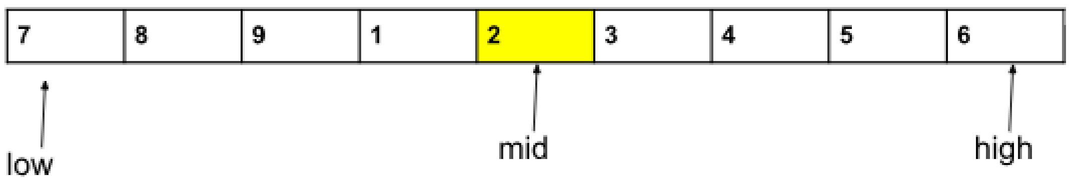
Here, we can easily observe, that we have to search in a sorted array. That is why, we can think of using the Binary Search algorithm to solve this problem.

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.

Observation:

To utilize the binary search algorithm effectively, it is crucial to ensure that the input array is sorted. By having a sorted array, we guarantee that each index divides the array into two sorted halves. In the search process, we compare the target value with the middle element, i.e. arr[mid], and then eliminate either the left or right half accordingly. This elimination becomes feasible due to the inherent property of the sorted halves(*i.e. Both halves always remain sorted*).

However, in this case, the array is both rotated and sorted. As a result, the property of having sorted halves no longer holds. This disruption in the sorting order affects the elimination process, making it unreliable to determine the target's location by solely comparing it with arr[mid]. To illustrate this situation, consider the following example:



target = 8,
Considering the comparison made, such as target > arr[mid] (e.g., 8 > 2), we would expect the target to be in the right half. However, due to the array rotation, the number 8 is actually situated in the left half. This rotation creates a challenge in the elimination process.

Key Observation: Though the array is rotated, we can clearly notice that for every index, one of the 2 halves will always be sorted. In the above example, the right half of the index mid is sorted.

So, to efficiently search for a target value using this observation, we will follow 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.

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 the index mid.
4. 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**.
5. Once, the 'mid' points to the target, the index will be returned.
6. This process will be inside a loop and the loop will continue until low crosses high. If no index is found, we will return -1.

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

▼ Code ›

```
def search(arr, n, k):
    low = 0
    high = n - 1
    while low <= high:
        mid = (low + high) // 2

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

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

if __name__ == "__main__":
    arr = [7, 8, 9, 1, 2, 3, 4, 5, 6]
    n = 9
    k = 1
    ans = search(arr, n, k)
    if ans == -1:
        print("Target is not present.")
    else:
        print("The index is:", ans)
```

Output: The index is: 3

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.**Reason:** We are using binary search to search the target.

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).