Description

33. Search in Rotated Sorted Array
81. Search in Rotated Sorted Array II

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., [0,0,1,2,2,5,6] might become [2,5,6,0,0,1,2]).

You are given a target value to search. If found in the array return true, otherwise return false.

```
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
```

Idea

The primitive idea is to brute force and enumerate all the elements and see if we can find the target, which take O(n) time. But we did not use the sorted property. To speed up, we may want to try O(logn) where we might want to use binary search to throw away half of the impossible candidates.

We can first assume no duplicate exists in the array (LC33) and discuss about the duplication later. Take [4,5,6,7,0,1,2] for example, we cut it into two halves and see which half we can throw away. We don't know where the pivot is, so there may be serveral circumstances we want to discuss. For this example, we have pivot at location between 7 and 0. We have a start and end index, the mid index is where we want to cut.

- 1. We cut before the pivot (before 0), then there might be two sub situations.
  - 1.1 The target is between start and mid, we would definitely know we can throw away

second half [mid,end] and search in [start, mid]
    - 1.2 Otherwise we can throw the first half [start,mid] and search in [mid, end]
- 2. We cut after the pivot (after 7), then there are other two sub situations.
    - 2.1 The target is between mid and end, we can throw away first half [start, mid] and search in [mid, end]
    - 2.2 Otherwise we can throw the second half [mid, end] and search in [start, mid]

And we can compare nums[mid] with the last element nums[nums.length - 1], if num[mid] > nums[nums.length - 1], we know we cut before the pivot. Because we can always throw away half of the remaining candidates, the time complexity is O(logn) with constant space. In the implementation, we can actually compare with nums[end] because the way how we throw away the impossible half make no difference, even though the meaning might be a little different.

Java

```
class Solution {

    public int search(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0;
        int end = nums.length - 1;

        while (start + 1 < end) {
            int mid = (end - start) / 2 + start;
            if (nums[mid] > nums[end]) {
                if (nums[start] <= target && target <= nums[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else {
                if (nums[mid] <= target && target <= nums[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }

        if (nums[start] == target) {
            return start;
        }
        if (nums[end] == target) {
```

```
            return end;
        }
        return -1;
    }
}
```

C++

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        if (nums.empty()) {
            return -1;
        }

        int start = 0;
        int end = nums.size() - 1;
        int last = nums[end];

        while (start + 1 < end) {
            int mid = (end - start) / 2 + start;
            if (nums[mid] > last) {
                if (nums[start] <= target && target <= nums[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else {
                if (nums[mid] <= target && target <= nums[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }

        return nums[start] == target ? start : nums[end] == target ? end : -1;
    }
};
```

For the follow up (LC81), we need to consider if there are duplicates in the input array. The only difference is that there might be one more situation where nums[mid] == nums[end]. And there is no way to tell whether we should throw away first half or second half. The only thing we are sure of is that we can throw away one element which nums[end], this way we can avoid infinite loop in the case where all input numbers are same. So we only need to add this one more condition

based on previouse implementation. So the average time is O(logn), but the worse case may be O(n) when input numbers are all the same. Still constant space.

Java

```
public boolean search(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return false;
        }

        int start = 0;
        int end = nums.length - 1;

        while (start + 1 < end) {
            int mid = (end - start) / 2 + start;
            if (nums[mid] > nums[end]) {
                if (nums[start] <= target && target <= nums[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else if (nums[mid] < nums[end]){
                if (nums[mid] <= target && target <= nums[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            } else {
                end--;
            }
        }

        return nums[start] == target || nums[end] == target;
    }
}
```

C++

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        if (nums.empty()) {
            return false;
        }

        int start = 0;
```

```
        int end =  nums.size() - 1;
        while (start + 1 < end) {
            int mid = (end - start) / 2 + start;
            if (nums[mid] > nums[end]) {
                if (nums[start] <= target && target <= nums[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else if (nums[mid] < nums[end]) {
                if (nums[mid] <= target && target <= nums[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            } else {
                end--;
            }
        }

        return nums[start] == target || nums[end] == target;
    }
};
```

Summary

- Throw away half of the impossible candidates at a time.
- Binary search, O(logn).
- Discuss different situations and code accordingly.
- Two pointers.