旋转有序数组搜索总结

1. LeetCode 33. Search in Rotated Sorted Array (在旋转有序数组中搜索)

题目：

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

(i.e., `[0,1,2,4,5,6,7]` might become `[4,5,6,7,0,1,2]`).

You are given a target value to search. If found in the array return its index, otherwise return `-1`.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

思路：

这道题让在旋转数组中搜索一个给定值，若存在返回坐标，若不存在返回 -1。

我们还是考虑二分搜索法，但是这道题的难点在于我们不知道原数组在哪旋转了，我们还是用题目中给的例子来分析，对于数组 [0 1 2 4 5 6 7] 共有下列七种旋转方法（红色表示中点之前或者之后一定为有序的）：

0 1 2 **4 5 6 7**

7 0 1 **2 4 5 6**

6 7 0 **1 2 4 5**

5 6 7 **0 1 2 4**

4 5 6 7 0 1 2

2 4 5 6 7 0 1

1 2 4 5 6 7 0

二分搜索法的关键在于获得了中间数后，判断下面要搜索左半段还是右半段，我们观察上面红色的数字都是升序的，由此我们可以观察出规律：

- 如果中间的数小于最右边的数，则右半段是有序的；
- 若中间数大于最右边数，则左半段是有序的；

我们只要在有序的半段里用首尾两个数组来判断目标值是否在这一区域内，这样就可以确定保留哪半边了，代码如下：

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) return mid;
            if (nums[mid] < nums[right]) {
                if (nums[mid] < target && nums[right] >= target) left = mid +
1;
                else right = mid - 1;
            } else {
                if (nums[left] <= target && nums[mid] > target) right = mid -
1;
                else left = mid + 1;
            }
        }
        return -1;
    }
};
```

看了上面的解法，你可能会产生个疑问，为啥非得用中间的数字跟最右边的比较呢？难道跟最左边的数字比较不行吗，当中间的数字大于最左边的数字时，左半段也是有序的啊，如下所示（蓝色表示中点之前一定为有序的）：

0 **1** **2** **4** 5 6 7

7 0 1 2 4 5 6

6 7 0 1 2 4 5

5 6 7 0 1 2 4

4 **5** **6** **7** 0 1 2

2 **4** **5** **6** 7 0 1

1 **2** **4** **5** 6 7 0

貌似也可以做，但是有一个问题，那就是在二分搜索中，nums[mid] 和 nums[left] 还有可能相等的，当数组中只有两个数字的时候，比如 [3, 1]，那么我们该去取那一边呢？

由于只有两个数字且 nums[mid] 不等于 target，那么 target 只有可能在右半边出现。那么最好的方法就是让其无法进入左半段，那么就需要左半段是有序的，而且由于一定无法同时满足 nums[left] <= target && nums[mid] > target，因为 nums[left] 和 nums[mid] 相等，同一个数怎么可能同时大于等于 target，又小于 target。

由于这个条件不满足，则直接进入右半段继续搜索即可，所以等于的情况要加到 nums[mid] > nums[left] 的情况中，变成大于等于，参见代码如下：

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) return mid;
            if (nums[mid] >= nums[left]) {
                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 -1;
    }
};
```

2. LeetCode 81. Search in Rotated Sorted Array II (在旋转有序数组中搜索之二)

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
```

Follow up:

- This is a follow up problem to Search in Rotated Sorted Array, where `nums` may contain duplicates.
- Would this affect the run-time complexity? How and why?

这道是之前那道 Search in Rotated Sorted Array 的延伸，现在数组中允许出现重复数字，这个也会影响我们选择哪半边继续搜索，由于之前那道题不存在相同值，我们在比较中间值和最右值时就完全符合之前所说的规律：**如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的**。

而如果可以有重复值，就会出现来面两种情况，[3 1 1] 和 [1 1 3 1]，对于这两种情况中间值等于最右值时，目标值3既可以在左边又可以在右边，那怎么办么，对于这种情况其实处理非常简单，只要把最右值向左一位即可继续循环，如果还相同则继续移，直到移到不同值为止，然后其他部分还采用 Search in Rotated Sorted Array 中的方法，可以得到代码如下：

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int n = nums.size(), left = 0, right = n - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target) return true;
            if (nums[mid] < nums[right]) {
                if (nums[mid] < target && nums[right] >= target) left = mid +
1;
                else right = mid - 1;
            } else if (nums[mid] > nums[right]){
                if (nums[left] <= target && nums[mid] > target) right = mid -
1;
                else left = mid + 1;
            } else --right;
        }
        return false;
    }
};
```