Topic Introduction

Welcome back, PrepLetter reader! Today's theme is a twist on a classic: **Binary Search in Rotated Sorted Arrays**.

Binary search is one of the most fundamental algorithms in coding interviews. At its core, it's a way to repeatedly halve your search space in a sorted array to quickly find a target value. Normally, this requires the array to be strictly sorted. But what if the array was sorted, then rotated at some unknown pivot? Now, the order is disrupted, and a vanilla binary search won't always work.

A quick recap: **Binary search** works by comparing the middle element of your search range to the target. If they match, you're done. If the target is smaller, you search the left half; if larger, the right half. This gives a time complexity of O(log n).

Why is binary search in rotated arrays important?
These problems test your ability to adapt a known, efficient algorithm to a slightly broken environment. They're very popular in interviews because they check your understanding of array properties, your ability to spot patterns, and how you tweak algorithms for edge cases (like duplicates or unknown pivots).

Example (not from the problems below):
Suppose you have [6, 7, 8, 1, 2, 3, 4, 5], which is a sorted array rotated at index 3 (the '1'). If you're looking for 3, normal binary search doesn't work directly, because the order "wraps around" the pivot. You'll need to reason about which half of the array is still sorted to decide where to search next.

Today's problems all use this core concept:

- **Search in Rotated Sorted Array:** Find a target in a rotated, *distinct* sorted array.
- **Find Minimum in Rotated Sorted Array:** Find the smallest value (i.e., the rotation point).
- **Search in Rotated Sorted Array II:** Like the first, but the array may have *duplicates*.

Why group these?
They all require adapting binary search to rotated arrays. The first tests your ability to find a target. The second helps you practice finding the rotation point (the minimum). The third adds complexity with duplicates, which can break some assumptions and require careful handling.

Let's dive in!

Problem 1: Search in Rotated Sorted Array

Problem Link

Problem Statement (in my words):
Given an array of distinct integers that was sorted in ascending order, then rotated at some unknown pivot, and a target value, return the index of the target if it exists. If not, return -1.

Example:
Input: nums = [4, 5, 6, 7, 0, 1, 2], target = 0
Output: 4
Why? The value '0' is at index 4.

Thought Process:

- Since the array was rotated, one half is always sorted.
- At every step, we can check which half is sorted and decide where to search next.
- This is a classic “modified binary search” problem.

Pen and Paper:

If this feels tricky, grab a sheet and draw the steps. Mark the mid, left, right pointers and see how the sorted halves shift as you search.

Try This:

Input: nums = [6, 7, 1, 2, 3, 4, 5], target = 3

What should the output be?

Brute-force Approach:

Loop through the array and check each element.

- Time: O(n)
- Space: O(1)

Optimal Approach: Modified Binary Search

- At each step, check if the left or right half is sorted.
- If the target is in the sorted half, search there.
- Otherwise, search the other half.

Step-by-Step:

- Set left and right pointers at the start and end of the array.
- While left <= right:
    - Compute mid index.
    - If nums[mid] == target, return mid.
    - If left half (nums[left] to nums[mid]) is sorted:
        - If target is in this range, search left half.
        - Else, search right half.
    - If right half is sorted:
        - If target is in this range, search right half.
        - Else, search left half.
- If not found, return -1.

Python Solution:

```
def search(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = (left + right) // 2

        if nums[mid] == target:
            return mid
```

```
        # Check if left half is sorted
        if nums[left] <= nums[mid]:
            # Target is in the left sorted part
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # Right half is sorted
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1
```

Time Complexity: O(log n)

Space Complexity: O(1)

Code Walkthrough:

- We keep shrinking our search space by half each time.
- The key is checking which half is sorted and if the target could be in that half.
- No extra space is used.

Trace of a Test Case:

nums = [4, 5, 6, 7, 0, 1, 2], target = 0

- left = 0, right = 6, mid = 3 (nums[3] = 7)
    - Left half [4,5,6,7] is sorted.
    - 0 is not in [4,7], so search right.
- left = 4, right = 6, mid = 5 (nums[5] = 1)
    - Right half [1,2] is sorted.
    - 0 is not in [1,2], so search left.
- left = 4, right = 4, mid = 4 (nums[4] = 0)
    - Found target at index 4.

Try This:

nums = [8, 9, 2, 3, 4], target = 4

What index will you get?

Try to solve this problem yourself before looking at the code above!

Problem 2: Find Minimum in Rotated Sorted Array

Problem Link

Problem Statement (in my words):

Given a rotated sorted array of distinct integers, find the minimum element.

How is this different?

Instead of looking for a specific value, we're always looking for the *smallest* value (the pivot point). The logic is similar: at each step, one half is sorted, and the minimum is in the unsorted half.

Example:

Input: nums = [3, 4, 5, 1, 2]

Output: 1

Brute-force Approach:

Loop through and find the smallest.

- Time: O(n)

Optimal Approach: Modified Binary Search

- The minimum element is the only element whose previous is greater than itself.
- If the array is already sorted (nums[left] < nums[right]), then nums[left] is the minimum.
- Otherwise, binary search for the inflection point where the order breaks.

Step-by-Step:

- Set left and right pointers.
- While left < right:
    - Compute mid.
    - If nums[mid] > nums[right] => minimum is to the right.
        - left = mid + 1
    - Else => minimum is at mid or to the left.
        - right = mid
- When left == right, nums[left] is the minimum.

Pseudocode:

```
function findMin(nums):
    left = 0
    right = len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[right]:
            left = mid + 1
        else:
            right = mid
    return nums[left]
```

Example Walkthrough:

nums = [3, 4, 5, 1, 2]

- left=0, right=4, mid=2 (nums[2]=5)
    - 5 > 2, so left = 3
- left=3, right=4, mid=3 (nums[3]=1)
    - 1 <= 2, so right = 3

- left=3, right=3 -> done. Minimum is nums[3]=1

Try This:

nums = [7, 8, 9, 2, 3, 4, 5, 6]

What is the minimum?

Time Complexity: O(log n)

Space Complexity: O(1)

Problem 3: Search in Rotated Sorted Array II

Problem Link

Problem Statement (in my words):

Given a rotated sorted array, which may contain duplicates, and a target value, determine if the target exists.

What's more challenging?

Duplicates make it harder to decide which half is sorted, since nums[left] == nums[mid] could happen. This weakens our ability to always choose a half confidently.

Example:

Input: nums = [2, 5, 6, 0, 0, 1, 2], target = 0

Output: True

Brute-force Approach:

Loop through and check each element.

- Time: O(n)

Optimal Approach (Modified Binary Search with Duplicates):

- If nums[left] == nums[mid] == nums[right], we can't tell which half is sorted. In that case, increment left and decrement right to skip duplicates.
- Otherwise, proceed as in Problem 1, checking which half is sorted and searching accordingly.

Pseudocode:

```
function search(nums, target):
    left = 0
    right = len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return True
        if nums[left] == nums[mid] == nums[right]:
            left += 1
            right -= 1
        elif nums[left] <= nums[mid]:   # left half is sorted
            if nums[left] <= target < nums[mid]:
                right = mid - 1
```

```
            else:
                left = mid + 1
        else:  # right half is sorted
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return False
```

Example Walkthrough:

nums = [2, 5, 6, 0, 0, 1, 2], target = 0

- left=0, right=6, mid=3 (nums[3]=0)
    - Found target

Try changing the target to 3. What happens?

Try This:

nums = [1, 1, 3, 1], target = 3

Can you trace the steps?

Time Complexity:

- O(log n) in the best case
- O(n) in the worst case (when duplicates force us to linearly scan chunks)

Space Complexity: O(1)

Summary and Next Steps

Today, we tackled three classic “rotated sorted array” problems, all united by their need for a modified binary search. You learned to:

- Identify which half of a rotated array is sorted.
- Use this insight to search for a target or the minimum value.
- Handle the extra complexity of duplicates, which can sometimes force a linear scan.

Key Patterns:

- Always compare nums[mid] to nums[left] and nums[right] to decide your next step.
- With duplicates, be cautious: if you can’t tell which half is sorted, skip duplicates at the ends.

Common Mistakes:

- Not updating left/right pointers correctly.
- Forgetting the possibility of all elements being the same (in Problem 3).
- Assuming O(log n) always holds, even with lots of duplicates.

Action List

- Solve all three problems yourself, even the one with code provided.
- For Problem 2 and 3, try solving them with a simple linear scan and compare the performance.
- Explore other problems that use binary search in creative ways (like “Find Peak Element” or “Search a 2D Matrix”).
- Compare your solution to others on LeetCode — pay attention to edge cases and code clarity.
- If you get stuck, slow down and draw out a few steps on paper. Practice will make the patterns click!

Keep at it — these patterns come up often and mastering them will make you much more confident in interviews!