

■ Search in Rotated Sorted Array

Binary search on a broken array — one half is always sorted!

● Binary Search

■ MEDIUM

■ What You Will Learn

- What 'rotation' means and how it breaks the array
- The key observation: one half is ALWAYS sorted
- How to decide which half to search
- The modified binary search pattern for rotated arrays

■ Intuition First

Take a sorted array `[0,1,2,4,5,6,7]` and rotate it 3 places. You get `[4,5,6,7,0,1,2]`. It's 'almost sorted' — split it at the mid, and ONE of the two halves is always perfectly sorted. We figure out which half is sorted, check if target is in that range, and eliminate the other half.

■ Key Observation

No matter how the array is rotated, when you pick a mid point, either the LEFT half (`arr[left]...arr[mid]`) OR the RIGHT half (`arr[mid]...arr[right]`) will be completely sorted. Use this to navigate!

■ Visual Explanation

```
Original:  [0, 1, 2, 4, 5, 6, 7]
Rotated:   [4, 5, 6, 7, 0, 1, 2]
           L         M         R

Left half:  [4,5,6,7] - SORTED (arr[L]=4 <= arr[M]=7)
Right half: [0,1,2]   - sorted too, but we check left first

Target = 0:
Is 0 in [4..7]? NO → search right half → L = M+1

Target = 5:
Is 5 in [4..7]? YES → search left half → R = M-1

Decision Rule:
if arr[L] <= arr[M]: left half is sorted
    if arr[L] <= target < arr[M]: go left
    else: go right
else: right half is sorted
    if arr[M] < target <= arr[R]: go right
    else: go left
```

■■ Step-by-Step Logic

1. Compute mid as usual
 2. Check if left half is sorted: **arr[left] <= arr[mid]**
 - If YES and target is in [arr[left], arr[mid]) → go left
 - Otherwise → go right
 3. Else right half is sorted: **arr[mid] < arr[right]**
 - If YES and target is in (arr[mid], arr[right]) → go right
 - Otherwise → go left
-

■ Java Code

```
1  class Solution {
2  public int search(int[] nums, int target) {
3  int left = 0, right = nums.length - 1;
4
5  while (left <= right) {
6  int mid = left + (right - left) / 2;
7
8  if (nums[mid] == target) return mid;
9
10 // Check if LEFT half is sorted
11 if (nums[left] <= nums[mid]) {
12 // Is target in the sorted left half?
13 if (nums[left] <= target && target < nums[mid]) {
14 right = mid - 1; // Go left
15 } else {
16 left = mid + 1; // Go right
17 }
18 } else {
19 // RIGHT half is sorted
20 // Is target in the sorted right half?
21 if (nums[mid] < target && target <= nums[right]) {
22 left = mid + 1; // Go right
23 } else {
24 right = mid - 1; // Go left
25 }
26 }
27 }
28 return -1;
29 }
30 }
```

■ Dry Run

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

```
Step 1: L=0, R=6, mid=3 → nums[3]=7 ≠ 0  
nums[L]=4 ≤ nums[mid]=7 → left sorted  
Is 0 in [4..7)? NO → L = mid+1 = 4
```

```
Step 2: L=4, R=6, mid=5 → nums[5]=1 ≠ 0  
nums[L]=0 > nums[mid]=1? No → left sorted: nums[4]=0 ≤ nums[5]=1  
Is 0 in [0..1)? YES → R = mid-1 = 4
```

```
Step 3: L=4, R=4, mid=4 → nums[4]=0 == 0 → return 4 ✓
```

■ Complexity

Complexity	Value	What it means
Time	$O(\log n)$	Still halving every step despite rotation
Space	$O(1)$	Only constant extra space

■ Common Mistakes

■ Mistake

Using strict $<$ instead of $\leq$ when checking if left is sorted ($\text{nums}[\text{left}] \leq \text{nums}[\text{mid}]$). The equal sign matters when $\text{left} == \text{mid}$ (single element).

■ Mistake 2

Forgetting the boundary conditions in range checks. Use $\text{nums}[\text{left}] \leq \text{target}$ AND $\text{target} < \text{nums}[\text{mid}]$ (note: $<$ not $\leq$ on mid side).

■ Quick Revision

✓	One half is always sorted — use this to decide where to search
✓	If $\text{nums}[\text{left}] \leq \text{nums}[\text{mid}]$: left half is sorted
✓	Check if target is in the sorted half's range
✓	If yes: search that half. If no: search the other
✓	Time $O(\log n)$, Space $O(1)$

■ Visual Thinking

Imagine a roller coaster that goes up, then has a sudden drop, then goes up again. At any midpoint on the track, one side of the car is always a smooth slope (sorted). You look at the slope, check if your destination is on it, and ride that side. This is the rotated array problem.