33. Search in Rotated Sorted Array

Binary Search

Problem Description

Medium Array

nums might have been rotated at some unknown pivot index k, causing the order of elements to change. After the rotation, the part of the array after the pivot index comes before the part of the array before the pivot index. Our goal is to find the index of a given integer target in the rotated array. If the target exists in the array, we should return its index; otherwise, return -1.

Since the array is rotated, a standard <u>binary search</u> won't immediately work. We need to find a way to adapt binary search to work under these new conditions. The key observation is that although the entire array isn't sorted, one half of the array around

In this problem, we have an integer array nums that is initially sorted in ascending order and contains distinct values. However,

work under these new conditions. The key observation is that although the entire array isn't sorted, one half of the array around the middle is guaranteed to be sorted.

The challenge is to perform this search efficiently, achieving a time complexity of $O(\log n)$, which strongly suggests that we should use binary search or a variation of it.

ntuition

The intuition behind the solution is to modify the binary search algorithm to handle the rotated array. We should focus on the

the end. Once we know which part is sorted, we can see if the target lies in that range. If it does, we adjust our search to stay

property that the rotation splits the array into two sorted subarrays. When we calculate the middle element during the binary search, we can determine which part of the array is sorted: the part from the start to the middle or the part from the middle to

within the sorted part. If not, we search in the other half.

To implement this, we have two pointers left and right that define the bounds of our search space. At each step, we compare the target with the midpoint to decide on which half to continue our search. There are four cases to consider:

1. If the target and the middle element both lie on the same side of the pivot (either before or after), we perform the standard binary search operation.

If the target is on the sorted side, but the middle element isn't, we search on the sorted side.
 If the target is not on the sorted side, but the middle element is, we search on the side that includes the pivot.
 If neither the target nor the middle element is on the sorted side, we again search on the side including the pivot.

- By repeatedly narrowing down the search space and focusing on either the sorted subarray or the subarray containing the pivot,
- we can find the target or conclude it's not present in O(log n) time.
- The solution implements a modified <u>binary search</u> algorithm to account for the rotated sorted array. Below is a step-by-step walkthrough of the algorithm as shown in the provided code snippet:

Initialize two pointers left and right to represent the search space's bounds. left starts at 0, and right starts at n - 1,

2. The binary search begins by entering a while loop that continues as long as left < right, meaning there is more than one

element in the search space.

found at that index in the array.

not sorted, the right-half must be sorted.

right half since the rotation must have happened there.

Since nums [mid] is equal to the target we return mid, which is 4.

10.

where n is the length of the input array nums.

right to mid to narrow the search space to this sorted part.

Solution Approach

3. Calculate mid using (left + right) >> 1. The expression >> 1 is equivalent to dividing by 2 but is faster, as it is a bitwise right shift operation.

Determine which part of the array is sorted by checking if nums [0] <= nums [mid]. This condition shows that the elements

- from nums[0] to nums[mid] are sorted in ascending order.

 5. Check if target is between nums[0] and nums[mid]. If it is, that means target must be within this sorted portion, so adjust
- 6. If the target is not in the sorted portion, it must be in the other half containing the rotation point. Update left to mid + 1 to shift the search space to the right half of the array.

If the sorted portion was the right half (nums[mid] < nums[n - 1]), check if target is between nums[mid] and nums[n - 1].

Adjust the search space accordingly by either moving left or right depending if the target is in the sorted portion or not.

- 8. This loop continues until the search space is narrowed down to a single element.
 9. After exiting the loop, check if nums[left] is the target and return left if that's the case, indicating that the target is
- runtime complexity as required by the problem statement.

Let's illustrate the solution approach with a small example. Assume we have the rotated sorted array nums = [6, 7, 0, 1, 2, 4,

5] and we are looking for the target value 1. The original sorted array before rotation might look something like [0, 1, 2, 4,

This binary search modification cleverly handles the rotation aspect by focusing on which part of the search space is sorted and

adjusting the search bounds accordingly. No additional data structures are used, and the algorithm strictly follows an O(log n)

1. We initialize our search bounds, so left = 0 and right = 6 (since there are 7 elements in the array).

5, 6, 7], and in this case, the pivot is at index 2, where the value 0 is placed in the rotated array.

Start the binary search loop. Since left < right (0 < 6), we continue.

If the element at nums [left] is not the target, return -1 to indicate that target is not found in the array.

Calculate the middle index, mid = (left + right) >> 1. This gives us mid = 3.
 Check if the left side is sorted by checking if nums[left] <= nums[mid] (comparing 6 with 1). It is false, so the left-half is

Since 1 (our target) is less than 6 (the value at nums [left]), we know the target is not in the left side. We now look into the

Check again where the sorted part is. Now nums[left] <= nums[mid] (1 <= 4) is true, which means we are now looking at</pre>

Check if the target is within the range of nums[left] and nums[mid]. Here, 1 is within 1 to 4, so it must be within this

Continue the loop. At this point, left = 4 and right = 5, mid = (4 + 5) >> 1 = 4. The value at nums[mid] is the target

6. Update left to mid + 1, which gives left = 4. 7. Re-evaluate mid in the next iteration of the loop. Now mid = (left + right) >> 1 = (4 + 6) >> 1 = 5. The middle element

at index 5 is 4.

the sorted part of the array.

we are looking for (1).

Solution Implementation

while left < right:</pre>

else:

else:

Python

Java

class Solution {

class Solution:

Example Walkthrough

range.

10. Now we update right to mid, setting right = 5.

- Using this approach, we successfully found the target 1 in the rotated sorted array and return the index 4. This example demonstrates the modified binary search algorithm used within the rotated array context.
- # Calculate the middle index
 mid = (left + right) // 2 # Updated to use floor division for clarity
 # Determine if the mid element is in the rotated or sorted part

if nums[0] <= target <= nums[mid]:</pre>

If target is between the first element and mid, go left

If target is between mid and last element, go right

Check if the left index matches the target, otherwise return -1

if nums[mid] < target <= nums[-1]: # Use -1 for last element index</pre>

def search(self, nums: List[int], target: int) -> int:

left, right = 0, len(nums) - 1

if nums[0] <= nums[mid]:</pre>

Else, go right

right = mid

left = mid + 1

return left if nums[left] == target else -1

public int search(int[] nums, int target) {

// Initialize start and end pointers.

int start = 0, end = arrayLength - 1;

int mid = (start + end) / 2;

if (nums[0] <= nums[mid]) {</pre>

end = mid;

start = mid + 1;

start = mid + 1;

end = mid;

} else {

} else {

const length = nums.length;

let rightIndex = length - 1;

} else {

} else {

} else {

class Solution:

while (leftIndex < rightIndex) {</pre>

// Use binary search to find the target

if (nums[0] <= nums[midIndex]) {</pre>

// Calculate the middle index

let leftIndex = 0;

} else {

// Binary search algorithm to find target.

// Calculate middle index of the current segment.

if (nums[0] <= target && target <= nums[mid]) {</pre>

// When middle element is on the rotated portion of the array.

if (nums[mid] < target && target <= nums[arrayLength - 1]) {</pre>

// When middle element is on the non-rotated portion of the array.

// Check if the target is also on the non-rotated portion and adjust end accordingly.

// Check if the target is also on the rotated portion and adjust start accordingly.

int arrayLength = nums.length;

// Length of the array.

while (start < end) {</pre>

Use binary search to find the target

Initialize the boundary indexes for the search

left = mid + 1
Else, go left
else:
right = mid

```
// After narrowing down to one element, check if it's the target.
        // If nums[start] is the target, return its index, otherwise return -1.
        return nums[start] == target ? start : -1;
C++
class Solution {
public:
    int search(vector<int>& nums, int target) {
        // Initialize the size of the input vector
        int size = nums.size();
        // Define the initial search range
        int left = 0, right = size - 1;
        // Perform binary search
        while (left < right) {</pre>
            // Find the middle index of the current search range
            int mid = left + (right - left) / 2; // Avoids potential overflow compared to (left + right) >> 1
            // Determine the side of the rotated sequence 'mid' is on
            if (nums[0] <= nums[mid]) {</pre>
                // 'mid' is in the left (non-rotated) part of the array
                if (nums[0] <= target && target <= nums[mid]) {</pre>
                    // Target is within the left (non-rotated) range, search left side
                    right = mid;
                } else {
                    // Search right side
                    left = mid + 1;
            } else {
                // 'mid' is in the right (rotated) part of the array
                if (nums[mid] < target && target <= nums[size - 1]) {</pre>
                    // Target is within the right (rotated) range, search right side
                    left = mid + 1;
                } else {
                    // Search left side
                    right = mid;
        // The final check to see if the target is found at 'left' index
        return (left == right && nums[left] == target) ? left : −1;
};
TypeScript
function search(nums: number[], target: number): number {
```

const midIndex = Math.floor((leftIndex + rightIndex) / 2); // shifted to use Math.floor for clarity

// Check if the first element is less than or equal to the middle element

// If target is between the first element and middle element

// If target is between the middle element and the last element

if (nums[midIndex] < target && target <= nums[length - 1]) {</pre>

if (nums[0] <= target && target <= nums[midIndex]) {</pre>

// Narrow down the right bound

// Target must be in the second half

// Target must be in the first half

rightIndex = midIndex;

leftIndex = midIndex + 1;

// Narrow down the left bound

leftIndex = midIndex + 1;

return nums[leftIndex] == target ? leftIndex : -1;

def search(self, nums: List[int], target: int) -> int:

Initialize the boundary indexes for the search

if nums[0] <= target <= nums[mid]:</pre>

rightIndex = midIndex;

// Check if we have found the target

left, right = 0, len(nums) - 1

while left < right:</pre>

else:

Use binary search to find the target

Calculate the middle index

if nums[0] <= nums[mid]:</pre>

Else, go right

right = mid

left = mid + 1 else: # If target is between mid and last element, go right if nums[mid] < target <= nums[-1]: # Use -1 for last element index left = mid + 1 # Else, go left else: right = mid # Check if the left index matches the target, otherwise return -1 return left if nums[left] == target else -1</pre> Time and Space Complexity Time Complexity

mid = (left + right) // 2 # Updated to use floor division for clarity

Determine if the mid element is in the rotated or sorted part

If target is between the first element and mid, go left

the loop, the time complexity of this operation is $O(\log n)$, where n is the number of elements in the array nums.

Space Complexity

The given code performs a binary search over an array. In each iteration of the while loop, the algorithm splits the array into half,

investigating either the left or the right side. Since the size of the searchable section of the array is halved with each iteration of

```
The space complexity of the algorithm is 0(1). The search is conducted in place, with only a constant amount of additional space needed for variables left, right, mid, and n, regardless of the size of the input array nums.
```