81. Search in Rotated Sorted Array II

Medium Array Binary Search

Problem Description

In this problem, we are given an array `nums` that has initially been sorted in non-decreasing order but then has been rotated around a pivot. The rotation means that `nums` is rearranged such that elements to the right of the pivot (including the pivot) are moved to the beginning of the array, and the remaining elements are shifted rightward. This rearrangement maintains the order of both subsets but not the entire array. Our task is to determine if a given `target` integer exists in `nums`. We need to accomplish this in an efficient way, aiming to minimize the number of operations performed.

Intuition

The challenge lies in the fact that due to the array's rotation, it's not globally sorted anymore—although within the two subarrays created by the rotation (one before and one after the pivot), the elements remain sorted. We can leverage this sorted property to apply a modified binary search to achieve an efficient solution.

Since the array is only partially sorted, a regular binary search isn't directly applicable, but we can modify our approach to work with the rotation. The key idea is to perform regular binary search steps, but at each step, figure out which portion of the array is sorted and then decide whether the `target` value lies within that sorted portion or the other portion.

We need to deal with the situation where the middle element is equal to the element on the right side of our searching range. This complicates things as it could represent the pivot point or a sequence of duplicate values. When such a case occurs, we can't make a definite decision about which part to discard, so we just shrink our search space by moving the right pointer one step left and continue the search.

By following this approach, we ensure that we are always halving our search space, leveraging the sorted nature of the subarrays whenever possible, and gradually converge towards the target element if it exists in the `nums` array.

Solution Approach

The algorithm uses a while-loop to repeatedly divide the search range in half, similar to a classic binary search, but with additional conditions to adapt it for the rotated array. The `nums` array is not passed by value. Instead, pointers (`l` and `r`) representing the left and right bounds of the current search interval are used to track the search space within the array, which is a space-efficient approach that doesn't involve additional data structures.

Here's a step-by-step breakdown of the code:

1. Set the initial `l` (left pointer) to `0` and `r` (right pointer) to `n - 1`, where `n` is the length of the `nums` array.
2. Start the `while` loop, which runs as long as `l < r`. This means we continue searching as long as our search space contains more than one element.
3. Calculate the mid-point `mid` using the expression `(l + r) >> 1`, which is equivalent to `(l + r) / 2` but faster computationally as it uses bit shifting.
4. Three cases are compared to determine the next search space:
   - **Case 1:** If `nums[mid] > nums[r]`, we know the left part from `nums[l]` to `nums[mid]` must be sorted. We then check if `target` lies within this sorted part. If it does, we narrow our search space to the left part by setting `r` to `mid`. Otherwise, `target` must be in the right part, and we update `l` to `mid + 1`.
   - **Case 2:** If `nums[mid] < nums[r]`, the right part from `nums[mid]` to `nums[r]` is sorted. If `target` is within this sorted interval, we update `l` to `mid + 1` to search in this part. Otherwise, we adjust `r` to the left by assigning it to `mid`.
   - **Case 3:** If `nums[mid] == nums[r]`, we can't determine the sorted part as the elements are equal, possibly due to duplicates. We can't discard any half, so we decrease `r` by one to narrow down the search space in small steps.
5. This process repeats, halving the search space each time until `l` equals `r`, at which point we exit the loop.
6. Check if the remaining element `nums[l]` is equal to `target`. If so, return `true`, else return `false`.

This solution leverages the sorted subarray properties induced by rotation and the efficiency of the binary search while handling the duplicates gracefully. This ensures that we minimize the search space as quickly as possible and determine the existence of the `target` in the array, thus decreasing the overall operation steps.

Example Walkthrough

Let's take a small example to illustrate the solution approach using the steps mentioned below:

Suppose `nums` is `[4, 5, 6, 7, 0, 1, 2]` and our `target` is `0`. This array is sorted and then rotated at the pivot element `0`.

1. Set the initial pointers: `l = 0`, `r = 6` (since there are 7 elements).
2. The `while` loop begins because `l < r` (`0 < 6`).
3. Calculate the mid-point: `mid = (l + r) >> 1`, hence `mid = 3`. The element at `mid` is `7`.
4. Proceed with comparing the three cases:
   - **Case 1:** Since `nums[mid]` (7) is greater than `nums[r]` (2), we find that the left part from `nums[l]` to `nums[mid]` is sorted. We check if `target` (0) could be in this sorted part. Given the sorted array [4,5,6,7], we see `target` is not there, so `l = mid + 1`, which makes `l = 4`.
   - The right bound `r` remains the same since `target` was not within the left sorted part.
5. Now `l = 4` and `r = 6`. We again calculate `mid = (4 + 6) >> 1`, so `mid = 5`. The element at mid is `1`.
   - **Case 2:** Since `nums[mid]` (1) is less than `nums[r]` (2), the right part is sorted ([1,2]). The `target` (0) is not within this interval, so we update `r` to `mid` which means `r = 5 - 1 = 4`.
   - The left bound `l` remains at 4 because the `target` was not located in the sorted right part.
6. We end up with `l = 4` and `r = 4` as both are equal, which means we exit the loop.
7. Check the remaining element `nums[l]`, which is `nums[4]` (0) against `target` (0). They match, so we would return `true`.

Through this example, the solution narrows down the search space by binary search while considering the effects of rotation. This results in an efficient way to determine if the `target` exists in the rotated sorted array, without having to search every element.

Python Solution

```python
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> bool:
        # The length of the input array
        num_length = len(nums)

        # Initialize the left and right pointers
        left, right = 0, num_length - 1

        # Binary search with modifications to handle the rotated sorted array
        while left < right:
            # Calculate the middle index
            mid = (left + right) // 2

            # If the middle element is greater than the rightmost element,
            # it means the smallest element is to the right of mid.
            if nums[mid] > nums[right]:
                # Target is in the left sorted portion
                if nums[left] <= target <= nums[mid]:
                    right = mid
                else:
                    left = mid + 1
            # If the middle element is less than the rightmost element,
            # it means the smallest element is to the left of mid.
            elif nums[mid] < nums[right]:
                # Target is in the right sorted portion
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid
            # If the middle element is equal to the rightmost element,
            # we can't determine the smallest element's position,
            # so we reduce the search space by one from the right.
            else:
                right -= 1

        # Final comparison to see if the target is at the left index
        return nums[left] == target
```

Java Solution

```java
class Solution {
    public boolean search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        // Continue searching while the window is valid
        while (left < right) {
            int mid = left + (right - left) / 2; // Avoid potential overflow of (left + right)

            // If middle element is greater than the rightmost element, the pivot is in the right half
            if (nums[mid] > nums[right]) {
                // If target lies within the left sorted portion
                if (nums[left] <= target && target <= nums[mid]) {
                    right = mid; // Narrow down to left half
                } else {
                    left = mid + 1; // Search in the right half
                }
            }

            // If middle element is less than the rightmost element, the left half is sorted properly
            else if (nums[mid] < nums[right]) {
                // If target lies within the right sorted portion
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1; // Narrow down to right half
                } else {
                    right = mid; // Search in the left half
                }
            }

            // If middle element equals the rightmost element, we can't determine the pivot
            // So we reduce the search space by moving the right pointer one step to the left
            else {
                right--;
            }
        }

        // After the loop ends, left == right,
        // checking if we have found the target
        return nums[left] == target;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        // Initialize the start and end indices
        int start = 0, end = static_cast<int>(nums.size()) - 1;

        // While the search space is valid
        while (start <= end) {
            // Calculate the midpoint index
            int mid = start + (end - start) / 2;

            // Check if the middle element is the target
            if (nums[mid] == target) {
                return true;
            }

            // When middle element is greater than the last element, it means
            // the left half is sorted correctly
            if (nums[mid] > nums[end]) {
                // Check if target is in the sorted half
                if (target >= nums[start] && target < nums[mid]) {
                    end = mid - 1; // Narrow search to the left half
                } else {
                    start = mid + 1; // Narrow search to the right half
                }
            // When middle element is less than the last element, it means
            // the right half is sorted correctly
            } else if (nums[mid] < nums[end]) {
                // Check if target is in the sorted half
                if (target > nums[mid] && target <= nums[end]) {
                    start = mid + 1; // Narrow search to the right half
                } else {
                    end = mid - 1; // Narrow search to the left half
                }
            // When middle element is equal to the last element, we don't have enough
            // information, thus reduce the size of search space from the end
            } else {
                end--;
            }
        }

        // After the while loop, if we haven't returned true, then target isn't present
        return false;
    }
};
```

Typescript Solution

```typescript
function search(nums: number[], target: number): boolean {
    let left = 0;
    let right = nums.length - 1;

    // Iterate as long as the left pointer is less than the right pointer
    while (left < right) {
        // Calculate the mid-point index
        const mid = Math.floor((left + right) / 2);

        // If the middle element is greater than the element at right,
        // the rotation is in the right half
        if (nums[mid] > nums[right]) {
            // Target is within the left sorted portion
            if (nums[left] <= target && target <= nums[mid]) {
                right = mid; // Narrow the search to the left half
            } else {
                left = mid + 1; // Narrow the search to the right half
            }
        // If the middle element is less than the element at right,
        // the rotation is in the left half
        } else if (nums[mid] < nums[right]) {
            // Target is within the right sorted portion
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1; // Narrow the search to the right half
            } else {
                right = mid; // Narrow the search to the left half
            }
        // If the middle element is equal to the element at right,
        // we are not sure where the rotation is
        } else {
            // We decrease the right pointer by one
            --right;
        }
    }

    // After the loop, if the left element is the target, return true,
    // otherwise, return false.
    return nums[left] === target;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given algorithm can primarily be considered as `O(log n)` in the case of a typical binary search scenario without duplicates because the function repeatedly halves the size of the list it's searching. However, in the worst-case scenario where the list contains many duplicates which are all the same as the target, the algorithm degrades to `O(n)` because the `else` clause where `r -= 1` could potentially be executed for a significant portion of the array before finding the target or determining it's not present.

Space Complexity

The space complexity of the code is `O(1)` because it uses a fixed number of variables, regardless of the input size. No additional data structures are used that would depend on the size of the input array.