154. Find Minimum in Rotated Sorted Array II

Hard | Array | Binary Search

Problem Description

The problem presents an array that has been sorted in ascending order but then has been rotated between 1 and n times. Rotating the array means moving the last element to the front. For example, if we have an array `[1, 2, 3, 4, 5]` and rotate it once, it becomes `[5, 1, 2, 3, 4]`. The task is to find the minimum element in this rotated array efficiently, even though the array may contain duplicates.

Intuition

Given that the array is initially sorted before the rotations, we can use binary search to find the minimum element. Here's the thought process:

- If the middle element is greater than the rightmost element, the smallest value must be to the right of the middle. This is because the elements are originally sorted in ascending order, so if there's a large number in the middle, the array must have been rotated at some point after that number.
- If the middle element is less than the rightmost element, the smallest value must be at the middle or to the left of the middle.
- If the middle element equals the rightmost element, we can't determine the position of the minimum element, but we can safely reduce the search space by ignoring the rightmost element since even if it is the minimum, it will exist as a duplicate elsewhere in the list.

The solution consistently halves the search interval, which means that the time complexity would be better than linear - in fact, it's O(log n) in the best case. However, when duplicates are present, and we encounter the same values at both the middle and end of our search range, we must decrease our search range in smaller steps, which in the worst case could result in an O(n) time complexity when all elements are duplicates. The approach is both practical and efficient for most cases.

Solution Approach

The solution uses a classic binary search pattern to find the minimum element. The key insight of binary search is to repeatedly divide the search interval in half. If the interval can be divided, it suggests that through a comparison in each step, one can eliminate half of the remaining possibilities. Here's how it is applied in this context:

- We start by setting two pointers, `left` at `0` and `right` at `len(nums) - 1`, to represent the search space's bounds.
- While `left` is less than `right`, we are not done searching:
  - We calculate `mid` by taking the average of `left` and `right`, effectively dividing the search space in half.
  - If `nums[mid] > nums[right]`, we know that the smallest value must be to the right of `mid`, so we set `left` to `mid + 1`.
  - If `nums[mid] < nums[right]`, we then know that the smallest value is either at `mid` or to its left. So, we move `right` to `mid`.
  - If `nums[mid]` is equal to `nums[right]`, we cannot be certain where the smallest value is, but we can reduce the search space by decrements `right` by `1`. This is because even if `nums[right]` is the smallest value, it will not be lost as the property of the array being sorted (aside from the rotation) means a duplicate must exist to the left.
- Once `left` equals `right`, the loop terminates and `left` (or `right`, since they are equal) points to the smallest element.

The code handles the presence of duplicates gracefully, ensuring the search space is narrowed even when the value of `nums[mid]` is equal to `nums[right]`. While binary search usually has an `O(log n)` complexity, this variant's complexity can degrade to `O(n)` in the worst-case scenario when there are many identical elements.

The reason why this approach is efficient is because it strives to minimize the number of elements considered at each step, allowing for an early exit in many cases as compared to a brute-force linear search that would always take `O(n)` time.

Example Walkthrough

Let's consider the rotated sorted array `nums = [4, 5, 6, 7, 0, 1, 2]` and walk through how the solution approach finds the minimum element.

1. Set the `left` pointer to `0` and the `right` pointer to `6`, since `len(nums) - 1 = 6`. Our array now looks like this: `4, 5, 6, 7, 0, 1, 2`, with `left` pointing to `4` and `right` pointing to `2`.
2. Since `left` is less than `right`, we calculate the middle index `mid = (left + right) // 2`, which is `(0 + 6) // 2 = 3`. The value at this index is `7`.
3. Now we compare the middle element `nums[mid]` with `nums[right]`:
   - `nums[mid]` is `7` and `nums[right]` is `2`. Since `7 > 2`, we know the smallest element is to the right of `mid`. We set `left` to `mid + 1`, which makes `left` `4`.
4. The array between indices `4` and `6` is now our search space: `0, 1, 2`. We repeat step 2 and find a new middle at `mid = (4 + 6) // 2 = 5`. The number at index `5` is `1`.
5. We now compare the `nums[mid]` with `nums[right]` again:
   - `nums[mid]` is `1` and `nums[right]` is `2`. Since `1 < 2`, we know the smallest element is at `mid` or to the left of `mid`. We move `right` to `mid`, making `right` `5`.
6. Our search space is now just `[0, 1]`. Calculating `mid` gives us `(4 + 5) // 2 = 4`. At `nums[mid]`, we have a `0`.
7. We compare `nums[mid]` with `nums[right]`:
   - `nums[mid]` is `0` and `nums[right]` is `1`. Since `0 < 1`, we continue to narrow the search and move `right` to `mid`, leaving us with `right` being `4`.
8. With both `left` and `right` pointing to the same index, which is `4`, the loop terminates. We have `left` equals `right` equals `4`, and `nums[left]` or `nums[right]` gives us the minimum element, which is `0`.

In this example, we successfully found the minimum element of the rotated sorted array using the binary search method, demonstrating the efficiency of the approach. The key takeaway is the decision-making at each step to reduce the search space, which is much quicker than a linear search especially when dealing with a larger array.

Solution Implementation

Python

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        # Initialize the left and right pointers to the start and end of the list respectively
        left, right = 0, len(nums) - 1

        # Continue searching while the left pointer is less than the right pointer
        while left < right:
            # Find the middle index by using bitwise right shift operation (equivalent to integer division by 2)
            mid = (left + right) >> 1

            # If the middle element is greater than the element at the right pointer...
            if nums[mid] > nums[right]:
                # ... the smallest value must be to the right of mid, so move the left pointer to mid + 1
                left = mid + 1
            # If the middle element is less than the element at the right pointer...
            elif nums[mid] < nums[right]:
                # ... the smallest value is at mid or to the left of mid, so move the right pointer to mid
                right = mid
            # If the middle element is equal to the element at the right pointer...
            else:
                # ... we can't be sure of the smallest, but we can reduce the search space by decrementing right pointer
                right -= 1

        # When the left pointer equals the right pointer, we've found the minimum, so return the element at left pointer
        return nums[left]
```

Java

```java
class Solution {
    public int findMin(int[] nums) {
        int left = 0; // Initialize the left boundary of the search
        int right = nums.length - 1; // Initialize the right boundary of the search

        // Perform a modified binary search
        while (left < right) {
            // Compute the middle index of the current search interval
            int mid = (left + right) >>> 1;

            // If the middle element is greater than the rightmost element,
            // the smallest value must be in the right part of the array.
            if (nums[mid] > nums[right]) {
                left = mid + 1;
            }
            // Else if the middle element is less than the rightmost element,
            // the smallest value must be in the left part of the array.
            else if (nums[mid] < nums[right]) {
                right = mid;
            }
            // If elements at mid and right are equal, we can't be sure of the smallest element's position,
            // but we can safely discard the rightmost element as the answer could still be to the left of it.
            else {
                right--;
            }
        }

        // After the loop, the left index will point to the smallest element in the rotated array
        return nums[left];
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    int findMin(std::vector<int>& nums) {
        // Initialize the search boundaries.
        int left = 0;
        int right = nums.size() - 1;

        // Continue searching as long as the left boundary is less than the right boundary.
        while (left < right) {
            // Find the middle index.
            int mid = left + (right - left) / 2; // Avoid potential overflow

            // If the middle element is greater than the right-most element,
            // the smallest value is to the right of mid; hence, update 'left'.
            if (nums[mid] > nums[right]) {
                left = mid + 1;
            }
            // If the middle element is less than the right-most element,
            // the smallest value is at mid or to the left of mid; hence, update 'right'.
            else if (nums[mid] < nums[right]) {
                right = mid;
            }
            // If the middle element is equal to the right-most element,
            // we can't decide the side of the minimum element, decrease 'right' to skip this duplicate.
            else {
                --right;
            }
        }

        // After the loop finishes, 'left' will point to the smallest element.
        return nums[left];
    }
};
```

TypeScript

```typescript
/**
 * Find the minimum value in a rotated sorted array.
 * The array may contain duplicates.
 * This function uses a binary search approach.
 * @param nums An array of numbers, rotated sorted order, possibly containing duplicates
 * @returns The minimum value found in the array
 */
function findMin(nums: number[]): number {
    // Initialize pointers for the binary search
    let leftIndex = 0;
    let rightIndex = nums.length - 1;

    // Perform binary search
    while (leftIndex < rightIndex) {
        // Calculate the middle index of the current search range
        const midIndex = leftIndex + Math.floor((rightIndex - leftIndex) / 2);

        // If the middle element is greater than the rightmost element, the minimum is to the right
        if (nums[midIndex] > nums[rightIndex]) {
            leftIndex = midIndex + 1;
        }
        // If the middle element is less than the rightmost element, the minimum is to the left or at midIndex
        else if (nums[midIndex] < nums[rightIndex]) {
            rightIndex = midIndex;
        }
        // If the middle element is equal to the rightmost element, we can't decide where the minimum is, move the right pointer left
        else {
            rightIndex--;
        }
    }

    // At the end of the loop, leftIndex is the smallest value
    return nums[leftIndex];
}
```


```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        # Initialize the left and right pointers to the start and end of the list respectively
        left, right = 0, len(nums) - 1

        # Continue searching while the left pointer is less than the right pointer
        while left < right:
            # Find the middle index by using bitwise right shift operation (equivalent to integer division by 2)
            mid = (left + right) >> 1

            # If the middle element is greater than the element at the right pointer...
            if nums[mid] > nums[right]:
                # ... the smallest value must be to the right of mid, so move the left pointer to mid + 1
                left = mid + 1
            # If the middle element is less than the element at the right pointer...
            elif nums[mid] < nums[right]:
                # ... the smallest value is at mid or to the left of mid, so move the right pointer to mid
                right = mid
            # If the middle element is equal to the element at the right pointer...
            else:
                # ... we can't be sure of the smallest, but we can reduce the search space by decrementing right pointer
                right -= 1

        # When the left pointer equals the right pointer, we've found the minimum, so return the element at left pointer
        return nums[left]
```


Time and Space Complexity

The provided code snippet is designed to find the minimum element in a rotated sorted array, handling duplicates. The algorithm employs a binary search technique.

Time Complexity:

The worst-case time complexity of this code is `O(n)`. In the average and best case, where the majority of elements are not duplicates, it approaches `O(log n)`. However, in the worst case, when the algorithm must decrement the `right` pointer one by one due to the presence of identical elements at the end of the array (`else: right -= 1`), the complexity degrades to `O(n)`. This happens when duplicates are present and cannot be ruled out by a regular binary search.

Space Complexity:

The space complexity of the code is `O(1)`. No additional space is utilized that is dependent on the input size; only a constant amount of extra space is used for variables like `left`, `right`, and `mid`.