153. Find Minimum in Rotated Sorted Array

Medium Array Binary Search

Problem Description

The problem presents an array containing a sequence of numbers in ascending order that has been rotated between 1 and n times. A rotation means that the elements are shifted to the right by one position, and the last element is moved to the first position. The task is to find the minimum element in this rotated array. The challenge is to achieve this with an algorithm that runs in O(log n) time, which suggests that a binary search approach should be used because binary search has a logarithmic time complexity and is typically applied to sorted arrays to find a specific element quickly.

Intuition

The key to solving this problem lies in understanding how a rotation affects a sorted array. Despite the rotation, a portion of the array remains sorted. If the array is not rotated or has been rotated a full cycle (n times), then the smallest element would be at the beginning. If it is rotated, the array is composed of two increasing sequences, and the minimum element is the first element of the second sequence.

Therefore, we can use binary search to quickly identify the point where the transition from the higher value to the lower value occurs, which indicates the smallest element. The binary search is modified here to compare the middle element with the first element and decide where to move next:

1. If the middle element is greater than the first element, the smallest value must be to the right of the middle element. Hence, we search the right half of the array.
2. If the middle element is less than the first element, then the smallest value is somewhere to the left of the middle element, or it could be the middle element itself. Here, we search the left half.

By applying this logic recursively to the halves, the point at which the smallest element exists can be found efficiently, satisfying the required time complexity.

Solution Approach

The implementation of the solution leverages a binary search algorithm to find the minimum element in the rotated array. Here are the detailed steps of the algorithm:

1. First, the solution checks if the first element of the array is less than or equal to the last element. If true, this indicates that the array is not rotated, or it is rotated a full cycle. Hence, the first element is already the minimum, and we can return it immediately.

   ```
   if nums[0] <= nums[-1]:
       return nums[0]
   ```

2. If the previous check fails, the solution sets two pointers, `left` and `right`, at the start and the end of the array, respectively. These pointers are used to dynamically narrow down the search region while performing the binary search.

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

3. The algorithm enters a loop that continues as long as the `left` pointer is less than the `right` pointer. The purpose of this loop is to repeatedly narrow the search space until the minimum element is identified.

   ```
   while left < right:
   ```

4. Inside the loop, the solution calculates the `mid` point between left and right pointers. This mid point is used to compare the elements and decide which half of the array to search next.

   ```
   mid = (left + right) >> 1
   ```

5. The next step is to compare the element at the `mid` index with the first element in the array. If `nums[mid]` is greater than `nums[0]`, we know that the smallest element must be to the right of `mid`, so we move the `left` pointer to `mid + 1`.

   ```
   if nums[0] <= nums[mid]:
       left = mid + 1
   ```

6. Otherwise, if `nums[mid]` is less than `nums[0]`, the minimum element is to the left of `mid`, or it could be `mid` itself, so we move the `right` pointer to `mid`.

   ```
   else:
       right = mid
   ```

7. The loop continues until the `left` and `right` pointers converge to the index of the minimum element. At this point, we can return `nums[left]` as the minimum element of the array.

   ```
   return nums[left]
   ```

This approach guarantees that the running time will be O(log n) because it repeatedly eliminates half of the search space in each iteration, which is characteristic of a binary search.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following rotated array:

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

Initially, we check if the array is not rotated or has been rotated a full cycle:

```
if nums[0] <= nums[-1]:
    return nums[0]
```

However, `nums[0]` is `4` and `nums[-1]` is `2`, so we do not return `nums[0]` because the array has been rotated.

Next, we set our `left` and `right` pointers:

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

Hence, `left = 0` and `right = 6`.

Entering the loop, we calculate our `mid` index:

```
mid = (left + right) >> 1
```

This gets us `mid = 3`, with `nums[mid] = 7`.

We now compare `nums[mid]` with `nums[0]`. Since `7` (middle element) is greater than `4` (the first element), we update the `left` to `mid + 1`:

```
left = mid + 1
```

Now, `left = 4` and `right = 6`.

On the next iteration, `mid = (4 + 6) >> 1 = 5`. The value `nums[mid]` is `1`, which is less than `nums[0]`.

Since `nums[mid]` is less than `nums[0]`, we update the `right` pointer to `mid`:

```
right = mid
```

Now, `left = 4` and `right = 5`.

Continuing with the loop, we calculate the next `mid`:

```
mid = (left + right) >> 1 = (4 + 5) >> 1 = 4
```

The value at `mid` index is `0`, which is less than `nums[0]`. So, we update the `right` pointer to `mid` again:

```
right = mid
```

Since now both `left` and `right` are `4`, the loop terminates, and we return the value `nums[left]` which is `0`, the smallest element in the rotated array:

```
return nums[left]
```

Thus, following the provided approach, we efficiently find the minimum element in a rotated sorted array in O(log n) time.

Python Solution

```python
from typing import List

class Solution:
    def findMin(self, nums: List[int]) -> int:
        # If the array is not rotated (or sorted in ascending order),
        # then the smallest element is the first element.
        if nums[0] <= nums[-1]:
            return nums[0]

        # Initialize the left and right pointers.
        left, right = 0, len(nums) - 1

        # Perform a binary search for the minimum element.
        while left < right:
            # Calculate the midpoint index
            mid = (left + right) // 2  # Using // for floor division in Python 3

            # If the element at the midpoint is greater than or equal
            # to the first element, then the minimum is to the right.
            if nums[0] <= nums[mid]:
                left = mid + 1
            else:
                # Otherwise, the minimum is to the left, so we reduce the right bound.
                right = mid

        # After the loop, left will point to the smallest element.
        return nums[left]

```

Java Solution

```java
class Solution {
    public int findMin(int[] nums) {
        int length = nums.length; // Store the length of the array for quick access

        // If the first element is less than or equal to the last element,
        // the minimum element must be at the starting index since the array is not rotated.
        if (nums[0] <= nums[length - 1]) {
            return nums[0];
        }

        // Initialize pointers for binary search
        int left = 0;
        int right = length - 1;

        // Conduct binary search to find the minimum element index
        while (left < right) {

            // Midpoint calculation
            int mid = left + (right - left) / 2;

            // Compare middle element with the first element to decide where to continue the search.
            // If nums[0] is less than or equal to nums[mid], the rotation index must be to the right
            // of mid, hence we adjust left to mid + 1.
            if (nums[0] <= nums[mid]) {
                left = mid + 1;
            } else {
                // If nums[0] is greater than nums[mid], the rotation index must be at mid or
                // to the left of mid, hence we adjust right to mid.
                right = mid;
            }
        }

        // After the search, left would be pointing at the minimum element in the rotated array.
        return nums[left];
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for using the vector data structure

class Solution {
public:
    // Function to find the minimum element in a rotated sorted array
    int findMin(std::vector<int>& nums) {
        int size = nums.size(); // Get the size of the vector

        // If the first element is less than or equal to the last element,
        // then the array is not rotated, so return the first element
        if (nums[0] <= nums[size - 1]) {
            return nums[0];
        }

        int left = 0; // Initialize left pointer to the start of the array
        int right = size - 1; // Initialize right pointer to the end of the array

        // Binary search to find the pivot, the smallest element
        while (left < right) {
            int mid = left + (right - left) / 2; // Find the mid index to prevent overflow

            // If the first element is less than or equal to the mid element,
            // then the smallest value must be to the right of mid
            if (nums[0] <= nums[mid]) {
                left = mid + 1;
            } else { // Otherwise, it is to the left of mid
                right = mid;
            }
        }

        // At this point, left is the index of the smallest element
        return nums[left];
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the minimum value in a rotated sorted array.
 * @param {number[]} nums - An array of numbers which has been rotated.
 * @returns {number} - The minimum value in the array.
 */
function findMin(nums: number[]): number {
    // Initialize two pointers for the start and end of the array segment.
    let start = 0;
    let end = nums.length - 1;

    // Use binary search to find the minimum element.
    while (start < end) {
        // Find the middle index by averaging start and end.
        const middle = (start + end) >>> 1;

        // Determine which part of the array to continue searching in.
        if (nums[middle] > nums[end]) {
            // If middle element is greater than end element,
            // the smallest value must be to the right of middle.
            start = middle + 1;
        } else {
            // Otherwise, the smallest value is to the left of middle, or at middle.
            end = middle;
        }
    }

    // When the while loop ends, start points to the smallest element.
    return nums[start];
}

```

Time and Space Complexity

The provided Python code implements a binary search algorithm to find the minimum element in a rotated sorted array. Here is the analysis of its time and space complexity:

Time Complexity:

The time complexity of the algorithm is `O(log n)`, where `n` is the length of the input list `nums`. This is because the algorithm uses a binary search approach, where it repeatedly divides the search interval in half. At each step, the algorithm compares the middle element with the boundary elements to determine which half of the array to search next. The number of steps required to find the minimum will, therefore, be proportional to the logarithm of the array size.

Space Complexity:

The space complexity of the algorithm is `O(1)`, as it uses only a constant amount of extra space. The variables `left`, `right`, and `mid` used for maintaining the bounds of the search space and no additional data structures are allocated that would depend on the size of the input list. Thus, the memory requirement remains constant irrespective of the input size.