[Click on the logo to find **Problem Statement**]

Intuition

The given code implements a binary search algorithm to find the minimum element in a rotated sorted array efficiently. The key intuition lies in leveraging the sorted nature of the array to determine the rotation point and locate the minimum element.

The algorithm maintains two pointers, `low` and `high`, representing the start and end of the search range. It iteratively checks if the subarray defined by these pointers is sorted. If it is, the minimum element is found at `nums[low]`, and the algorithm returns it.

If the array is rotated, the algorithm calculates the middle index, `mid`, and compares the elements at `low` and `mid`. If `nums[low]` is greater than `nums[mid]`, it indicates that the rotation point lies in the left half of the array. Consequently, the `high` pointer is updated to `mid`. Otherwise, the rotation point is in the right half, and the `low` pointer is set to `mid + 1`.

Approach

1. **Initialize Pointers:** Set two pointers, `low` and `high`, initially pointing to the start and end of the array, respectively.
2. **Binary Search Loop:** Enter a while loop that continues until the `low` pointer is greater than `high`.
3. **Check Sorted Subarray:** Inside the loop, check if the subarray defined by `low` and `high` is sorted. If it is, return the element at the `low` pointer as it represents the minimum element in the rotated array.
4. **Calculate Midpoint:** If the subarray is not sorted, calculate the midpoint (`mid`) of the current range.
5. **Identify Rotation Point:** Compare the elements at `low` and `mid`. If `nums[low]` is greater than `nums[mid]`, it suggests that the rotation point, and consequently the minimum element, lies in the left half of the current range. Update the `high` pointer to `mid`.
6. **Adjust Pointers:** If the rotation point is not in the left half, it must be in the right half. Update the `low` pointer to `mid + 1`.

7. **Final Result:** Continue the binary search until the `low` pointer is greater than `high` . The minimum element is then found at `nums[low]` .

Complexity

- Time complexity: O(log n)
- Space complexity: O(1)

Code

```
class Solution:
    def findMin(self, nums: List[int]) -> int:
        low, high = 0, len(nums) - 1
        while low <= high:
            if nums[low] <= nums[high]:
                return nums[low]
            mid = ((high - low) // 2) + low
            if nums[low] > nums[mid]:
                high = mid
            else:
                low = mid + 1
```

If you want to see more solutions to coding problems, you can visit:

GitHub