153. Find Minimum in Rotated Sorted Array

**Binary Search** 

Array

Medium

# **Problem Description**

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 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

of the second sequence.

Solution Approach

middle element itself. Here, we search the left half.

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

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

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.

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:

## 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

immediately. if nums[0] <= nums[-1]: return nums[0]

If the previous check fails, the solution sets two pointers, left and right, at the start and the end of the array, respectively.

Inside the loop, the solution calculates the mid point between left and right pointers. This mid point is used to compare the

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

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

These pointers are used to dynamically narrow down the search region while performing the binary search.

while left < right:</pre>

mid = (left + right) >> 1

elements and decide which half of the array to search next.

return nums [left] as the minimum element of the array.

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

iteration, which is characteristic of a binary search.

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]:</pre> left = mid + 1

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

```
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.
```

right = mid

else:

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

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

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

The loop continues until the left and right pointers converge to the index of the minimum element. At this point, we can

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

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

return nums[0]

**Example Walkthrough** 

Next, we set our left and right pointers: left, right = 0, len(nums) - 1

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

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

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

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

+ 1:

left = mid + 1

right = mid

right = mid

Hence, left = 0 and right = 6.

Entering the loop, we calculate our mid index:

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

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:

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

Solution Implementation

from typing import List

if nums[0] <= nums[-1]:</pre>

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

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

return nums[0]

while left < right:</pre>

return nums[left]

public int findMin(int[] nums) {

return nums[0];

while (left < right) {</pre>

} else {

return nums[left];

\* Finds the minimum value in a rotated sorted array.

\* @returns {number} - The minimum value in the array.

// Use binary search to find the minimum element.

const middle = (start + end) >>> 1;

if (nums[middle] > nums[end]) {

// Find the middle index by averaging start and end.

function findMin(nums: number[]): number {

let end = nums.length - 1;

while (start < end) {</pre>

**}**;

**/**\*\*

\*/

**TypeScript** 

let start = 0;

// Midpoint calculation

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

left = mid + 1;

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

// of mid, hence we adjust left to mid + 1.

if (nums[0] <= nums[length - 1]) {</pre>

Continuing with the loop, we calculate the next mid:

# then the smallest element is the first element.

# Perform a binary search for the minimum element.

# Initialize the left and right pointers.

# Calculate the midpoint index

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

the rotated array: return nums[left]

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

class Solution: def findMin(self, nums: List[int]) -> int: # If the array is not rotated (or sorted in ascending order),

**Python** 

left = mid + 1else: # Otherwise, the minimum is to the left, so we reduce the right bound. right = mid

mid = (left + right) // 2 # Using // for floor division in Python 3

int length = nums.length; // Store the length of the array for quick access

// the minimum element must be at the starting index since the array is not rotated.

// Compare middle element with the first element to decide where to continue the search.

// If nums[0] is greater than nums[mid], the rotation index must be at mid or

// If nums[0] is less than or equal to nums[mid], the rotation index must be to the right

// If the first element is less than or equal to the last element,

// Conduct binary search to find the minimum element index

// to the left of mid, hence we adjust right to mid.

// At this point, left is the index of the smallest element

\* @param {number[]} nums - An array of numbers which has been rotated.

// Initialize two pointers for the start and end of the array segment.

// Determine which part of the array to continue searching in.

minimum will, therefore, be proportional to the logarithm of the array size.

# If the element at the midpoint is greater than or equal

# to the first element, then the minimum is to the right.

# After the loop, left will point to the smallest element.

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

class Solution {

Java

```
right = mid;
        // After the search, left would be pointing at the minimum element in the rotated array.
        return nums[left];
C++
#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]) {</pre>
            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) {</pre>
            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]) {</pre>
                left = mid + 1;
            } else { // Otherwise, it is to the left of mid
                right = mid;
```

```
// 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];
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]:</pre>
            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:</pre>
           # 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]:</pre>
               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]
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:**

**Space Complexity:** 

The time complexity of the algorithm is  $0(\log n)$ , 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

The space complexity of the algorithm is 0(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.