1150. Check If a Number Is Majority Element in a Sorted Array

Easy | Array | Binary Search | Leetcode Link

Problem Description

Given an integer array `nums` which is sorted in non-decreasing order, and an integer `target`, the task is to determine whether `target` is a "majority" element in `nums`. A majority element is one that appears more than `nums.length / 2` times. The function should return `true` if `target` is indeed a majority element, and `false` otherwise.

Intuition

The intuition behind the solution comes from the property of the array being sorted in non-decreasing order. We can use binary search to quickly find the first and last occurrences of the `target` element. In Python, this can be efficiently done using the `bisect_left` and `bisect_right` functions from the `bisect` module.

- `bisect_left` returns the index of the first occurrence of `target` in `nums` (or the index where `target` would be inserted to maintain the sorted order if it's not present).
- `bisect_right` returns the index of the first element greater than `target` (which would be one past the last occurrence of `target` if `target` is in `nums`).

By subtracting the index returned by `bisect_left` from the index returned by `bisect_right`, we get the total number of times `target` appears in `nums`. If this number is greater than `nums.length / 2`, then `target` is a majority element, and we return `true`. If not, we return `false`.

Using binary search makes the solution very efficient even for large arrays, since we avoid scanning the whole array and operate with a time complexity of O(log n).

Solution Approach

The solution uses a binary search approach to find the first and last occurrences of the target element in the sorted array. The binary search algorithm is a well-known method that operates by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half; otherwise, reduce it to the upper half. Repeatedly checking in this manner until the value is found or the interval is empty.

Here's how the `bisect_left` and `bisect_right` functions contribute to the solution:

- `bisect_left(nums, target)`: This line of code uses the `bisect_left` function from Python's `bisect` module. Given the sorted array `nums`, it finds the leftmost position at which `target` should be inserted in order to maintain the sorted order. If `target` is already in `nums`, `bisect_left` will return the index of the first occurrence of `target`. This is effectively the start index of `target` in the array.

- `bisect_right(nums, target)`: Similarly, `bisect_right` finds the rightmost position to insert `target` while keeping the array sorted. If `target` exists in the array, `bisect_right` will return the index directly after the last occurrence of `target`. This is essentially the index at which `target` would no longer appear in the array.

With the indices from `bisect_left` and `bisect_right`, the code calculates the number of times `target` appears in the array by subtracting the left index from the right index (`right - left`). This gives us the total count of `target` in `nums`.

To determine if `target` is a majority element, the code compares the count of `target` with half of the array's length (`len(nums) // 2`). The integer division by two ensures that we have a threshold which `target`'s count must exceed to be considered a majority element. If the count is greater than this threshold, the function returns `true`; otherwise, it returns `false`.

The data structure used here is the list `nums`, and the algorithm implemented is binary search through the use of `bisect_left` and `bisect_right`. No additional data structures are necessary. This approach is efficient because it minimizes the number of elements inspected, and the binary search is performed in O(log n) time complexity, where n is the number of elements in `nums`.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the array `nums` and the `target` given as follows:

```
nums = [1, 2, 2, 3, 3, 3, 3]
target = 3
```

The array `nums` is sorted, and we want to determine whether `3` is a majority element. The majority element must appear more than `len(nums) / 2 = 7 / 2 = 3.5` times. Since the array length is 7, the target must appear more than 3 times to be a majority element.

Let's apply the binary search approach using the `bisect_left` and `bisect_right` functions from the `bisect` module:

1. Find the left index for the target `3` using `bisect_left`:

   ```
   from bisect import bisect_left
   left_index = bisect_left(nums, target) # left_index is 3
   ```

   This indicates that the first occurrence of `3` in the array `nums` is at index 3.

2. Find the right index for the target `3` using `bisect_right`:

   ```
   from bisect import bisect_right
   right_index = bisect_right(nums, target) # right_index is 7
   ```

   This suggests that the index directly after the last appearance of `3` in the array `nums` is 7.

3. Now we calculate the total count of `target` by subtracting the left index from the right index:

   ```
   count = right_index - left_index # count is 4
   ```

   The variable `count` now holds the total number of times `target` appears in `nums`, and in this case, it is 4.

4. Finally, check if `count` is greater than `len(nums) / 2` to determine if `target` is a majority element:

   ```
   is_majority = count > len(nums) // 2 # is_majority is True
   ```

   Since `4` is greater than `3.5`, we can confirm that `3` is indeed a majority element in the array `nums`.

So, using this binary search approach, we have determined that the `target` element `3` is a majority element in the array with minimal computation compared to traversing the entire array. The example validates the solution's ability to efficiently solve the given problem.

Python Solution

```python
from bisect import bisect_left, bisect_right

class Solution:
    def isMajorityElement(self, nums: List[int], target: int) -> bool:
        # Find the leftmost index where `target` should be inserted to keep the list sorted.
        left_index = bisect_left(nums, target)

        # Find the rightmost index where `target` should be inserted to keep the list sorted.
        right_index = bisect_right(nums, target)

        # Check if the count of `target` in the list is greater than half the length of the list.
        # This is done by comparing the difference between `right_index` and `left_index`, which
        # gives the number of occurrences of `target`, to half the length of the list.
        return right_index - left_index > len(nums) // 2
```

Java Solution

```java
class Solution {
    // Function to check if the target is the majority element in the sorted array
    public boolean isMajorityElement(int[] nums, int target) {
        // Find the start index of the target value
        int startIndex = findFirstOccurrence(nums, target);
        // Find the start index of the value immediately after the target
        int endIndex = findFirstOccurrence(nums, target + 1);

        // Check if the count of the target value is more than half of the array's length
        return (endIndex - startIndex) > nums.length / 2;
    }

    // Helper function to find the first occurrence of a value using binary search
    private int findFirstOccurrence(int[] nums, int value) {
        int left = 0;
        int right = nums.length;
        while (left < right) {
            // Compute the middle index
            int mid = left + (right - left) / 2;

            // Narrow down to the left half if the middle element is greater than or equal to the value
            if (nums[mid] >= value) {
                right = mid;
            } else {
                // Otherwise, narrow down to the right half
                left = mid + 1;
            }
        }
        // Return the starting index where the target value would be or is located
        return left;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Required for std::lower_bound and std::upper_bound

class Solution {
public:
    bool isMajorityElement(vector<int>& nums, int target) {
        // Use lower_bound to find the first occurrence of `target`
        auto firstOccurrence = std::lower_bound(nums.begin(), nums.end(), target);

        // Use upper_bound to find the position immediately after the last occurrence of `target`
        auto lastOccurrence = std::upper_bound(nums.begin(), nums.end(), target);

        // Calculate the number of times `target` appears in the vector
        int count = lastOccurrence - firstOccurrence;

        // Check if the count of `target` is more than half the size of the vector
        bool isMajority = count > nums.size() / 2;

        return isMajority;
    }
};
```

Typescript Solution

```typescript
// This function determines if a given target is the majority element in a sorted array.
function isMajorityElement(nums: number[], target: number): boolean {
    // Helper function that performs a binary search to find the start
    // index of a given number (x) in the sorted array.
    const binarySearch = (x: number): number => {
        let leftIndex = 0;
        let rightIndex = nums.length;
        // Perform a binary search.
        while (leftIndex < rightIndex) {
            let midIndex = (leftIndex + rightIndex) >> 1; // Equivalent to Math.floor((leftIndex + rightIndex) / 2)
            if (nums[midIndex] >= x) {
                rightIndex = midIndex;
            } else {
                leftIndex = midIndex + 1;
            }
        }
        return leftIndex;
    };

    // Using the helper function to find the first occurrence of the target.
    const firstTargetIndex = binarySearch(target);
    // Finding the first index past the last occurrence of the target
    // using the next number (target + 1).
    const firstIndexPastTarget = binarySearch(target + 1);

    // Determine if the target is the majority element by comparing the
    // number of occurrences to more than half the size of the array.
    return firstIndexPastTarget - firstTargetIndex > nums.length >> 1; // Equivalent to Math.floor(nums.length / 2)
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is determined by the functions `bisect_left` and `bisect_right` from Python's `bisect` module. Both functions perform binary search to find the leftmost and rightmost positions of `target` in the sorted array `nums`, respectively.

The binary search algorithm has a time complexity of `O(log n)`, where `n` is the number of elements in the array. Since the code performs two binary searches, one for `bisect_left` and one for `bisect_right`, the total time complexity is:

```
2 * O(log n) = O(log n)
```

This simplifies to `O(log n)` because the constants are dropped in Big O notation.

Space Complexity

The space complexity of the code is `O(1)` since it uses only a fixed amount of extra space. The variables `left` and `right` are used to store the indices found by the binary search, and no additional data structures are created that depend on the size of the input array `nums`. Therefore, the space requirements of the algorithm do not scale with the input size.