2529. Maximum Count of Positive Integer and Negative Integer

Easy | Array | Binary Search | Counting

Leetcode Link

Problem Description

In this problem, we are provided with a sorted array `nums` which is in non-decreasing order. Our task is to find the maximum count of either positive integers or negative integers present in the array. It is important to note that the number `0` is considered neither positive nor negative. We need to return the higher count between the total number of positive integers and the number of negative integers in the array.

Intuition

Since the array is sorted in non-decreasing order, all negative numbers, if any, will be positioned at the beginning of the array, followed by zeroes and then positive numbers. Identifying the number of positive numbers can be done by finding the first occurrence of a number greater than or equal to `1`. The count of positive numbers is then the total length of the array minus the index of this first positive number. Similarly, the number of negative numbers is the index where the first non-negative number (which could be `0`) is located, as this index is equal to the count of negative numbers before it.

We can efficiently find these points of transition from negative to zero and from zero to positive using a binary search technique. The Python `bisect` library offers functions like `bisect_left` which can be leveraged for this purpose:

- `bisect_left(nums, 1)` will return the index of the first positive integer.
- `bisect_left(nums, 0)` will give the index of the first non-negative integer, which is also the count of the negative integers.

Finally, we take the maximum between the count of positive numbers and negative numbers and return it as the solution. The use of binary search here allows the solution to be efficient even for large arrays.

Solution Approach

The solution approach takes advantage of the sorted nature of the input array and binary search algorithm to efficiently find the count of negative and positive numbers. It uses the `bisect` library in Python, which provides a collection of tools for handling sorted lists.

Here's a step-by-step breakdown of how the algorithm works:

1. Calculate the count of positive numbers by finding the index of the first positive integer. Since the array is sorted, all positive integers are at the end. The function `bisect_left(nums, 1)` finds the leftmost value greater than or equal to `1`. This is essentially the index of the first positive number. The count of positive numbers is then the length of the array minus this index.
2. Find the count of negative numbers by locating the index of the first non-negative integer (either `0` or the first positive). `bisect_left(nums, 0)` will give us this index directly, which represents the total number of negative integers since they are all to the left of this point in the sorted array.
3. The `max(a, b)` function is then used to compare the counts of positive numbers `(a)` and negative numbers `(b)` and return the larger of the two. This maximum value represents the largest group, either positive or negative numbers within the array.

The code uses `bisect_left` twice on the sorted array, making the time complexity `O(log n)`, where `n` is the size of the array `nums`. The space complexity is `O(1)` because we are only storing two integers for the counts and the result, using no additional space proportionate to the input size.

In essence, the implementation leverages binary search to pinpoint the transition points within the array—the first transition from negative to either zero or positive and the second transition from zero to positive—and then uses simple arithmetic and the `max` function to determine the answer.

Example Walkthrough

Let's consider the following example array of sorted integers:

```
nums = [-4, -3, -2, 0, 2, 3, 5]
```

In this example, we shall walk through the algorithm steps based on the solution approach.

1. **Find the count of positive numbers:**
   - We want to locate the index of the first positive integer. Since the array is non-decaying, we can use `bisect_left(nums, 1)` to find this point efficiently.
   - Applying `bisect_left(nums, 1)` to our example, we identify that the first occurrence of a number greater than or equal to `1` is positioned at index `4` where the value is `2`.
   - The count of positive integers is then the total array length `7` minus the index `4`, which equals `3`. So there are `3` positive numbers.
2. **Determine the count of negative numbers:**
   - Here, we aim to locate the index of the first non-negative number (`0` or a positive integer). `bisect_left(nums, 0)` will help find that index.
   - In our example, `bisect_left(nums, 0)` returns index `3` as the first location where the value `0` or a number greater does not precede it.
   - This indicates that there are precisely `3` negative integers before the zero found at index `3`.
3. **Choose the maximum count:**
   - We need to decide which group is larger: negative numbers or positive numbers. By using the `max(a, b)` function where `a` is the number of positives and `b` is the number of negatives, we can quickly infer the larger count.
   - Comparing `3` (positive count) and `3` (negative count), we see that they are equal. Therefore, the function should return `3`.

With these steps, the algorithm determines that the maximum count of either positive or negative integers in the array `nums` is `3`. The efficient use of binary search allows us to conclude without having to iterate over the entire array, thus maintaining a time complexity of `O(log n)`.

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def maximumCount(self, nums: List[int]) -> int:
        # Calculate the number of elements equal or greater than 1 using binary search.
        # bisect_left returns the leftmost place in the sorted list to insert the number 1
        # Subtract this from the length of the array to find the count of elements >= 1
        count_ones_or_more = len(nums) - bisect_left(nums, 1)

        # Calculate the number of elements which are negative using binary search.
        # bisect_left returns the leftmost place in the sorted list to insert number 0
        # It gives us the number of negative numbers since the list is sorted.
        count_negative = bisect_left(nums, 0)

        # Return the maximum of the two counts calculated
        return max(count_ones_or_more, count_negative)

```

Java Solution

```java
class Solution {

    // Method calculates the maximum count of either 0's or 1's in a sorted binary array
    public int maximumCount(int[] nums) {
        // Find the number of 1's by subtracting the index of the first 1 from the array length
        int countOfOnes = nums.length - firstOccurrence(nums, 1);
        // Find the first occurrence index of 0, which is also the count of 0's
        int countOfZeros = firstOccurrence(nums, 0);
        // Return the max count between 0's and 1's
        return Math.max(countOfOnes, countOfZeros);
    }

    // Helper method to find the first occurrence index of 'x' in the sorted array 'nums'
    private int firstOccurrence(int[] nums, int x) {
        int left = 0;
        int right = nums.length;
        // Binary search to find the first occurrence of 'x'
        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to (left + right) / 2 but faster
            // If mid element is greater than or equal to x, we move the right boundary
            if (nums[mid] >= x) {
                right = mid;
            } else {
                // If mid element is less than x, we move the left boundary
                left = mid + 1;
            }
        }
        // 'left' will point to the first occurrence of 'x' or nums.length if 'x' is not found
        return left;
    }
}

```

C++ Solution

```cpp
// Include the necessary header files
#include <vector>
#include <algorithm> // Needed for std::lower_bound function

class Solution {
public:
    // Function to find the maximum count of elements greater than or equal to 1
    // or the number of elements before the first occurrence of 1 in the sorted array.
    int maximumCount(std::vector<int>& nums) {
        // Find the number of elements that are greater than or equal to 1.
        // This is done by finding the lower bound of 1 in the array,
        // which gives an iterator to the first element not less than 1,
        // and then calculating the distance from this iterator to the end of the array.
        int countOfOnesOrGreater = nums.end() - std::lower_bound(nums.begin(), nums.end(), 1);

        // Similar to the above, find the lower bound of 0, which is the first occurrence of 0,
        // and then calculate the distance from the beginning of the array to this iterator.
        // This is the number of elements that are less than 1.
        int countBeforeFirstOne = std::lower_bound(nums.begin(), nums.end(), 0) - nums.begin();

        // Return the maximum of the two counts calculated above.
        return std::max(countOfOnesOrGreater, countBeforeFirstOne);
    }
};

```

Typescript Solution

```typescript
// Function that returns the maximum count of zeroes or ones by either
// counting the number of zeroes from the start or the number of ones from the end
function maximumCount(nums: number[]): number {
    // Helper function to perform a binary search for the target value (0 or 1)
    // and return the index of the first occurrence of the target in the sorted array
    const binarySearch = (target: number): number => {
        let left = 0;    // Starting index of the search range
        let right = nums.length; // Ending index of the search range (exclusive)

        // While the search range is not empty
        while (left < right) {
            // Calculate the midpoint to divide the search range
            const mid = (left + right) >>> 1; // Equivalent to Math.floor((left + right) / 2)

            // Narrow down the search range based on the comparison with target
            if (nums[mid] < target) {
                left = mid + 1; // Target must be in the upper half of the range
            } else {
                right = mid;    // Target is in the lower half or at the midpoint
            }
        }
        // Return the final index where the target should be inserted
        return left;
    };

    // Find the index of the first occurrence of 0 and 1 using binary search
    const indexZero = binarySearch(0);
    const indexOne = binarySearch(1);

    // Calculate the maximum count of either zeroes or ones
    // by choosing the higher count between the two
    return Math.max(indexZero, nums.length - indexOne);
}

```

Time and Space Complexity

Time Complexity

The given code consists mainly of two binary searches, one to find the index of the first occurrence of `1` using `bisect_left(nums, 1)` and another one to find the index of the first occurrence of `0` using `bisect_left(nums, 0)`. Since binary search has a time complexity of `O(log n)` for a list of `n` elements, and we are performing two binary searches, the overall time complexity of the function is `O(log n)`.

Space Complexity

The implemented function does not create any additional data structures that grow with the input size. Thus, the space complexity is constant, `O(1)`, regardless of the input size of `nums`.