1608. Special Array With X Elements Greater Than or Equal X

Easy Array Binary Search Sorting [Leetcode Link]

Problem Description

You are provided with an array called `nums` that contains non-negative integers. The array is classified as **special** if you can find a number `x` which meets a particular condition. The condition is that the count of numbers within the array `nums` that are greater than or equal to `x` must be exactly equal to `x`. It is important to note that `x` doesn't need to be a part of the `nums` array.

Your task is to find and return the number `x` if the array meets the criteria of being special. In the case where the array is not special, you should return `-1`. It is also mentioned that if an array is special, the value of `x` that satisfies the condition is always unique.

Intuition

The core idea to solve this problem efficiently is based on the realization that a sorted array makes it easier to find the count of elements greater than or equal to a given value. To elaborate, once `nums` is sorted, you can determine how many numbers are greater than or equal to `x` by finding the position of the first number in `nums` that is at least `x` and then calculating the number of elements after it in the array.

Here's how the thinking progresses towards a solution:

1. Sort the array. In Python, this can be achieved using `nums.sort()`.
2. Iterate through potential `x` values, starting from `1` to the size of the array `n`. For each potential `x`, use a binary search to find the leftmost position where `x` could be inserted into the array without violating the sorted order. This operation tells us the count of elements greater than or equal to `x` by subtracting the insertion index from the length of the array `n`. In Python, this can be achieved using `bisect_left(nums, x)` which is imported from the `bisect` module.
3. For each `x`, if the count of elements greater than or equal to `x` is equal to `x` itself, we have found the special value and return it.
4. If none of the `x` values meet the condition, then the array is not special, and we return `-1`.

By using the sorted array and binary search, we can determine the count of elements >= `x` quickly for each `x`, allowing us to find the special value or determine that it does not exist with a good time efficiency.

Solution Approach

The implementation of the solution uses several straightforward steps following an algorithmic pattern that takes advantage of array sorting and binary search - a common pattern when dealing with ordered datasets.

Let's break down the solution step-by-step:

1. **Sorting**: The input array `nums` is sorted in non-decreasing order. This is done to leverage the ordered nature of the array in subsequent steps which is essential for binary search. In Python, we achieve this using the `nums.sort()` method which sorts the list in place.

2. **Binary Search**: To find out how many numbers are greater than or equal to a number `x`, we can perform a binary search to find the index of the first number in `nums` that is equal to or greater than `x`. The `bisect_left` function from the `bisect` module is used here which takes a sorted list and a target value `x`, then finds the leftmost insertion point for `x` in the list. The use of `bisect_left` ensures we have an efficient O(log n) lookup for the index.

3. **Loop Over Potential `x` Values**: We know `x` can be at most the length of the array `n`. The solution iterates `x` from `1` through to `n` inclusive, checking if any of these values satisfy the special condition.

4. **Counting Greater or Equal Elements**: For each `x`, the solution calculates the number of elements greater than or equal to `x`. This is done using `n - bisect_left(nums, x)`. The `bisect_left` function gives us the index at which `x` could be inserted to maintain the list's sorted order. Therefore, the count of numbers greater than or equal to `x` is the length of `nums` minus this index.

5. **Verification and Return**: The loop checks whether each `x` value equals the count of elements greater than or equal to `x`. When it finds a match `(cnt == x)`, it returns `x` because we've found the unique number that makes the array special. If no such `x` is found by the time the loop ends, the solution returns `-1`, indicating that the array is not special.

The pattern followed here is a classic example of combining sorting with binary search to optimize the lookup steps, common in many algorithmic problems for reducing time complexity.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Suppose our given `nums` array is `[3, 6, 7, 7, 0]`.

1. **Sorting**: First, we need to sort the array `nums`. After sorting, it becomes `[0, 3, 6, 7, 7]`.

2. **Binary Search**: We will use binary search to find the position for potential `x` values. For example, if we're checking for `x = 3`, we want to find the index where `3` could be inserted.

3. **Loop Over Potential `x` Values**: We start checking for all potential `x` values starting from `1` up to the length of the array, which is `5` in this case. So, our potential `x` values are `1, 2, 3, 4, 5`.

4. **Counting Greater or Equal Elements**: We calculate the count of elements greater than or equal to `x` for each `x`. For instance:
   - For `x = 1`, binary search (`bisect_left`) insertion index in sorted `nums` is `0`. The count is `5 - 0 = 5`.
   - For `x = 2`, binary search insertion index is `1`. The count is `5 - 1 = 4`.
   - For `x = 3`, binary search insertion index is `1`. The count is `5 - 1 = 4`.
   - For `x = 4`, binary search insertion index is `2`. The count is `5 - 2 = 3`.
   - For `x = 5`, binary search insertion index is `5`. The count is `5 - 5 = 0`.
5. **Verification and Return**: We check each `x` against the count of elements greater or equal to it:
   - For `x = 1`, `5 != 1`. No match.
   - For `x = 2`, `4 != 2`. No match.
   - For `x = 3`, `4 != 3`. No match.
   - For `x = 4`, `count = 3`, and `x = 4` do not match. No match.
   - For `x = 5`, `count = 0`, and `x = 5` do not match. No match.

Since none of the potential `x` values resulted in the count being equal to `x` itself, we return `-1`. Therefore, the example array `[3, 6, 7, 7, 0]` is not special according to the problem statement.

Python Solution

```python
from bisect import bisect_left

class Solution:
    def specialArray(self, nums: List[int]) -> int:
        # Sort the input array.
        nums.sort()

        # Find the length of the nums array and store it in a variable n.
        n = len(nums)

        # Iterate through potential special values (x).
        for x in range(1, n + 1):
            # Use binary search (bisect_left) to find the leftmost position in nums
            # where x could be inserted, then subtract it from n to get the count
            # of elements greater than or equal to x.
            count_greater_or_equal_to_x = n - bisect_left(nums, x)

            # Check if count is equal to x (which is our definition of a special array).
            if count_greater_or_equal_to_x == x:
                # If it is a special array, return x.
                return x

        # If no special value is found, return -1.
        return -1

```

Java Solution

```java
class Solution {
    public int specialArray(int[] nums) {
        // Sort the array to enable binary search
        Arrays.sort(nums);
        int n = nums.length; // Get the length of the sorted array

        // Iterate through possible values of x
        for (int x = 1; x <= n; ++x) {
            int left = 0; // Initialize left pointer of binary search
            int right = n; // Initialize right pointer of binary search

            // Perform binary search to find the first position where the value is >= x
            while (left < right) {
                int mid = (left + right) >> 1; // Calculate the middle index
                if (nums[mid] >= x) {
                    // If mid value is >= x, shrink the right end of the search range
                    right = mid;
                } else {
                    // If mid value is < x, shrink the left end of the search range
                    left = mid + 1;
                }
            }

            // Calculate the count of numbers >= x
            int countGreaterOrEqualX = n - left;

            // If the count of numbers >= x equals x, we found the special value
            if (countGreaterOrEqualX == x) {
                return x; // Return the special value of x
            }
        }

        // If no special value is found, return -1
        return -1;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int specialArray(vector<int>& nums) {
        // Calculate the size of the array
        int size = nums.size();

        // Sort the array in non-decreasing order
        sort(nums.begin(), nums.end());

        // Iterate for each potential special value 'x' starting from 1 to size of array
        for (int x = 1; x <= size; ++x) {
            // Calculate the count of numbers greater than or equal to 'x' using lower_bound
            // which returns an iterator to the first element that is not less than 'x'.
            // Subtracting this from the beginning of the array gives the number of elements
            // less than 'x', and subtracting from 'size' gives the elements greater than or equal to 'x'.
            int count = size - (lower_bound(nums.begin(), nums.end(), x) - nums.begin());

            // If the number of elements greater than or equal to 'x' is exactly 'x',
            // Then we found the special value 'x' and return it
            if (count == x) {
                return x;
            }
        }

        // If no such 'x' exists, return -1
        return -1;
    }
};

```

Typescript Solution

```typescript
// Function to find if there exists any integer x such that x is the number of elements
// in nums that are greater than or equal to x. If such an x is found, return x, otherwise return -1.
function specialArray(nums: number[]): number {
    // Total number of elements in the array
    const length = nums.length;
    // Left bound of the binary search
    let lowerBound = 0;
    // Right bound of the binary search, cannot be more than the length of the array
    let upperBound = length;

    // Perform binary search
    while (lowerBound < upperBound) {
        // Calculate the middle index of the current search range
        const mid = Math.floor((lowerBound + upperBound) / 2);
        // Calculate the count of numbers that are greater than or equal to mid
        const count = nums.reduce((accumulator, value) => accumulator + (value >= mid ? 1 : 0), 0);

        // If count equals mid, we found a special number, so return it
        if (count === mid) {
            return mid;
        }

        // If count is greater than mid, we need to search in the upper half of the range
        if (count > mid) {
            lowerBound = mid + 1;
        } else {
            // If count is less than mid, we need to search in the lower half of the range
            upperBound = mid;
        }
    }

    // If we exit the loop without finding a special number, return -1
    return -1;
}

```

Time and Space Complexity

The provided Python code attempts to find a special array with a non-negative integer `x`. An array is special if the number of numbers greater than or equal to `x` is equal to `x`.

Time Complexity

The time complexity of the given code consists of two parts:

1. Sorting the `nums` array.
2. Performing a binary search for each value of `x` in the sorted `nums` array using the `bisect_left` function.

The sorting operation on an array of `n` elements has a time complexity of `O(n log n)`.

The loop runs from `1` to `n+1` times; in each iteration, a binary search is performed using the `bisect_left` function. The binary search has a time complexity of `O(log n)` for each search.

Considering that the binary search is performed `n` times, the total time complexity for all binary searches in the worst case is `O(n log n)`.

Hence, the overall time complexity is dominated by the sorting and the `n` binary searches, which in combination yields a time complexity of `O(n log n + n log n)`. This simplifies to `O(n log n)` since the sorting term is the dominant term and the additional `n log n` term does not change the asymptotic growth rate.

Space Complexity

The space complexity of the algorithm is determined by the space used besides the input array `nums`.

1. Sorting is in-place, so it does not use additional space proportional to the input array.
2. The binary search uses only a few variables such as `x` and `cnt`, which take up constant space.

There is no additional space that is dependent on the size of the input, thus the space complexity is `O(1)`, which means it uses constant additional space.

In summary:

- Time Complexity: `O(n log n)`
- Space Complexity: `O(1)`