2996. Smallest Missing Integer Greater Than Sequential Prefix Sum

Easy Array Hash Table Sorting

Problem Description

You are given an array of integers called `nums`, which starts at index 0. We consider a "sequential prefix" of this array to be a section of the array starting at index 0 and continuing up to some index `i`. This sequential prefix is only considered sequential if every element after the first (from `nums[1]` to `nums[i]`) is exactly 1 greater than the element that comes before it. Even just the first element `nums[0]` by itself counts as a sequential prefix.

The task is to find the smallest integer `x` that is not present in the array `nums`, with the condition that `x` must be equal to or greater than the sum of the elements in the longest sequential prefix of the array. In essence, we're looking for the smallest missing number that's also higher than or equal to this particular sum.

Intuition

The intuition behind the solution begins with identifying the sum of the longest sequential prefix of the array. Since the array is 0-indexed, we start with the first element (the sum `s` starts as `nums[0]`) and check the subsequent elements to see if they continue the sequence by incrementing by 1.

Once we reach an element that breaks this rule (not one greater than its predecessor), or we reach the end of the array, we stop. At this point, we have the sum `s` of the longest sequential prefix.

The next step is to find the smallest integer `x` not in the array but also greater than or equal to our sum `s`. To do this efficiently, we can use a hash table (or `set` in Python) that allows us to quickly check if an integer is part of the `nums` array or not. We start iterating from our calculated sum `s` and check each consecutive integer. The moment we hit an integer not in our hash table, we have found our `x`, and that integer is our answer.

By utilizing the hash table, we avoid iterating over the `nums` array repeatedly, which would increase the time complexity significantly. Instead, we get a fast lookup and can thus solve the problem more efficiently.

Solution Approach

The solution approach to the given problem could be broken down into the following algorithmic steps:

1. **Initialization**: We start by initializing the sum `s` to the first element of the array `nums[0]`. We also set a pointer `j` to `1` as we will start checking for the continuity of the sequential prefix from the second element.
2. **Finding the Longest Sequential Prefix**: We enter a while loop that continues as long as `j` is less than the length of `nums` and the current element `nums[j]` is exactly 1 more than the previous element `nums[j - 1]`. Inside this loop, we keep adding `nums[j]` to our sum `s` and increment `j` after each iteration.
3. **Hash Table Creation**: After finding the sum of the longest sequential prefix, we prepare for the missing integer search by creating a set (acting as a hash table) called `vis` which includes all the elements of `nums`. This is done so that we can later check in constant time whether an element is present in `nums` or not.
4. **Finding the Missing Integer**: With the hash table ready, we iterate through integers starting from `s` using a counter. For each value `x`, we check if `x` is not present in the set `vis`. The first integer `x` which is not found in the set is the smallest missing integer that is also greater than or equal to the sum `s` of the longest sequential prefix. As soon as we find such an `x`, it is returned as the result.

The algorithm exploits the properties of a hash table (in Python, a set) to ensure that the check if an integer is a part of the `nums` array happens in constant time. This is a critical step in ensuring that the algorithm is efficient even when dealing with large arrays.

Moreover, the use of a counter iteration starting from `s` is a simple and effective way to search for the missing integer, since it guarantees that we will eventually find the smallest integer not in the array and also meet the condition of being greater than or equal to `s`.

Here's a reference to the actual implementation from the solution code and the approach:

```
class Solution:
    def missingInteger(self, nums: List[int]) -> int:
        s, i = nums[0], 1
        while i < len(nums) and nums[j] == nums[j - 1] + 1:
            s += nums[j]
            i += 1
        vis = set(nums)
        for x in count(s):
            if x not in vis:
                return x
```

In the code snippet, the `count` function effectively iterates from `s` indefinitely until the condition for the missing integer is met.

Example Walkthrough

Let's illustrate the solution approach with a simple example. Suppose we have the following array of integers:

```
nums = [1, 2, 3, 5, 6]
```

First, we initialize `s = nums[0]`, which is `1` in this case, and set a pointer `j` to `1` to start checking the continuity from the second element.

Now, we iterate through the array to find the longest sequential prefix:

- `nums[1]` is `2`, which is `nums[0] + 1`. So, we add `2` to `s`, making `s = 3`, and increment `j` to `2`.
- `nums[2]` is `3`, which is `nums[1] + 1`. We add `3` to `s`, now `s = 6`, and increment `j` to `3`.
- `nums[3]` is `5`, which is not `nums[2] + 1`. The sequence is broken here, so we stop the iteration.

The sum `s` of the longest sequential prefix is `6`.

Next, we create a set `vis` that contains all elements from `nums` to facilitate constant time checks for the presence of an element in the array.

We then start iterating from `6` (which is `6`) to find the smallest missing integer `x` that is also greater than or equal to `s` and not in `nums`.

- We check `6` first, but `6` is in `vis`. So, we move to the next integer, `7`.
- `7` is not present in `vis`, which means it is the smallest missing integer fulfilling the conditions.

Therefore, the function would return `7` as the answer for this input array.

Solution Implementation

Python

```
from typing import List
from itertools import count

class Solution:
    def missingInteger(self, nums: List[int]) -> int:
        # Initialize sum with the first element and start index at 1
        cumulative_sum, index = nums[0], 1

        # Loop to calculate the sum of consecutive integers
        while index < len(nums) and nums[index] == nums[index - 1] + 1:
            cumulative_sum += nums[index]
            index += 1

        # Create a set for nums, it would help in fast look-up
        visited_numbers = set(nums)

        # This starts a count from the sum of consecutive numbers
        # until it finds an integer not present in nums
        for missing_number in count(cumulative_sum):
            if missing_number not in visited_numbers:
                return missing_number
```

Java

```
class Solution {
    public int missingInteger(int[] nums) {
        // Initialize sum with the first element of the array
        int sum = nums[0];

        // Calculate the sum of consecutive numbers starting with nums[0]
        // Stop if the current number isn't consecutive to the previous one
        for (int i = 1; i < nums.length && nums[i] == nums[i - 1] + 1; ++i) {
            sum += nums[i];
        }

        // Create a boolean array to track visited numbers up to 50
        boolean[] visited = new boolean[51];

        // Fill the visited array, marking each number in 'nums' as visited
        for (int num : nums) {
            visited[num] = true;
        }

        // Start checking for the missing integer starting from the calculated sum
        for (int i = sum; ++i) {
            // Return the first unvisited number or the number beyond the range of visited
            if (i >= visited.length || !visited[i]) {
                return i;
            }
        }
    }
}
```

C++

```
#include <vector>
#include <bitset>

class Solution {
public:
    int missingInteger(vector<int>& nums) {
        // Initialize sum with the first integer in the array.
        int sum = nums[0];

        // This loop adds up a sequence of consecutive integers starting from the first element,
        // stopping when a non-consecutive number is found.
        for (int i = 1; i < nums.size() && nums[i] == nums[i - 1] + 1; ++i) {
            sum += nums[i];
        }

        // Initialize a bitset where its size is one more than the maximum value we expect to check.
        // This is a compact way to store whether each number is present or not.
        bitset<51> visited;

        // Fill the bitset by setting the bit corresponding to each number in the 'nums' vector to 1 (true).
        for (int num : nums) {
            visited[num] = 1;
        }

        // Starting from the sum (the smallest possible missing number in a sequence),
        // incrementally check if each number is missing by checking the bitset.
        // Notes: The condition 'x < 51' is technically not needed if 'nums' only contains numbers in valid range.
        for (int x = sum; x < 51; ++x) {
            // If a number is not visited (i.e., the corresponding bit is 0), it's missing from the sequence.
            if (!visited[x]) {
                // Return the first missing number encountered.
                return x;
            }
        }

        // In case all numbers up to 50 are present, just return 51.
        return 51;
    }
};
```

TypeScript

```
function missingInteger(nums: number[]): number {
    // Start by initializing the sum 's' with the first number in the array
    let sum = nums[0];

    // Iterate through the array, checking if the current number is consecutive
    for (let i = 1; i < nums.length && nums[i] === nums[i - 1] + 1; ++i) {
        // Add the current number to the sum if it's the consecutive number
        sum += nums[i];
    }

    // Create a Set to efficiently check the existence of numbers in 'nums'
    const visitedNumbers: Set<number> = new Set(nums);

    // Start looking for the missing integer, incrementally checking each number greater than the sum
    for (let x = sum; ; ++x) { // Infinite loop, will break internally when the missing integer is found
        if (!visitedNumbers.has(x)) { // Check if the number 'x' is not in the 'visitedNumbers'
            // Found the missing integer, breaking the loop and returning 'x'
            return x;
        }
    }
}
```

```
from typing import List
from itertools import count

class Solution:
    def missingInteger(self, nums: List[int]) -> int:
        # Initialize sum with the first element and start index at 1
        cumulative_sum, index = nums[0], 1

        # Loop to calculate the sum of consecutive integers
        while index < len(nums) and nums[index] == nums[index - 1] + 1:
            cumulative_sum += nums[index]
            index += 1

        # Create a set for nums, it would help in fast look-up
        visited_numbers = set(nums)

        # This starts a count from the sum of consecutive numbers
        # until it finds an integer not present in nums
        for missing_number in count(cumulative_sum):
            if missing_number not in visited_numbers:
                return missing_number
```

Time and Space Complexity

Time Complexity

The given Python code consists primarily of a while-loop and a for-loop with an embedded set membership check.

- The while-loop runs while `j` is less than the length of the `nums` array and each consecutive element in `nums` is one more than the previous element. In the worst case, this loop runs in `O(n)` time, where `n` is the length of the `nums` array, because it can iterate through the entire array in sequence.
- After the while-loop, the `set()` operation that creates `vis` is `O(n)`, because it must visit each element of the `nums` list once.
- The for-loop that begins with `for x in count(s):` is a bit tricky because it relies on what is missing from the `nums` array. However, it will execute at most `n` times before finding a missing integer that is not in `vis`, leading to a `O(n)` time complexity for this segment as well. The `count` function from the `itertools` library generates an iterator starting from the sum `s`. Each iteration checks whether `x` is not in `vis`, which is a `O(1)` operation because `vis` is a set.

Therefore, the total time complexity of the code is `O(n)`, because all steps are linear in the size of the input array `nums`.

Space Complexity

The space complexity of the code comes from the additional data structures used:

- The `vis` variable is a set that can contain at most `n` unique integers from the `nums` array, giving it `O(n)` space complexity.
- There are a fixed number of scalar variables (`s` and `j`), which use a constant amount of space, `O(1)`.
- The `count` function generates an iterator which doesn't consume additional space proportional to `n`, since it is just a number generator.

Thus, the overall space complexity of the code is `O(n)`, dominated by the size of the `vis` set.