1909. Remove One Element to Make the Array Strictly Increasing

Easy Array Leetcode Link

Problem Description

Given an array `nums` of integers, you need to determine if there is one element which can be removed to make the array strictly increasing. An array is strictly increasing if each element is greater than the previous one (`nums[i] > nums[i - 1]` for all `i` from 1 to `nums.length - 1`). The goal is to return `true` if the array can be made strictly increasing by removing exactly one element, otherwise, return `false`. It's also important to note that if the array is already strictly increasing without any removals, the answer should be `true`.

Intuition

The solution approach involves two key observations:

1. If we encounter a pair of elements where the current element is not greater than its predecessor (`nums[i] <= nums[i-1]`), it presents a potential violation of the strictly increasing condition.
2. To resolve this violation, we have two choices: either remove the current element (`nums[i]`) or the previous element (`nums[i-1]`). After making the removal, we should check if the rest of the array is strictly increasing.

The function `check(nums, i)` takes care of evaluating whether the array `nums` becomes strictly increasing if the element at index `i` is removed. It iterates through the array and skips over the index `i`. As it iterates, it maintains a `prev` value that stores the last valid number in the sequence. If `prev` becomes greater than or equal to the current number in the sequence at any point, that means the sequence is not strictly increasing, so it returns `false`. If it finishes the loop without finding such a scenario, it means the sequence is strictly increasing, and it returns `true`.

With these observations in hand, the main portion of the code starts checking elements from the beginning of the array. When it finds a violation where `nums[i - 1] >= nums[i]`, it knows it's time to check the two possibilities: removing `nums[i - 1]` or `nums[i]`. It calls the `check` function for these two indices and returns `true` if either of these checks returns `true`, reflecting that the array could indeed be made strictly increasing by removing one of those elements.

Solution Approach

The Python code provided defines a `Solution` class with a method `canBeIncreasing`, which takes an integer list `nums` as input and returns a boolean indicating whether the array can become strictly increasing by removing exactly one element.

Here's a step-by-step walkthrough of the implementation:

1. A helper function `check(nums, i)` is defined, which takes the array `nums` and an index `i`. This function is responsible for checking if the array can be strictly increasing by ignoring the element at index `i`. To do that:
   - It initializes a variable `prev` to `-inf` (negative infinity) to act as the comparator for the first element (since any integer will be greater than `-inf`).
   - It then iterates over all the elements in `nums` and skips the element at the index `i`. For each element `num`, it checks if `prev` is greater than or equal to `num`. If this condition is true at any point, it means removing the element at index `i` does not make the array strictly increasing, so it returns `false`.
   - If it completes the loop without finding any such violations, the function returns `true`, indicating that ignoring the element at index `i` results in a strictly increasing array.
2. Within the `canBeIncreasing` method, a loop commences from the second element (`i` starts at 1) and compares each element with its predecessor.
   - As long as the elements are in strictly increasing order (`nums[i - 1] < nums[i]`), the loop continues.
   - When a non-increasing pair is found, the code checks two cases by invoking the `check` function: one where `nums[i - 1]` is ignored (by passing `i - 1`) and one where `nums[i]` is ignored (by passing `i`).
3. The result of the method is the logical OR between these two checks:
   - `check(nums, i - 1)` confirms if the sequence is strictly increasing by ignoring the pre-violation element.
   - `check(nums, i)` confirms if the sequence is strictly increasing by ignoring the post-violation element.
   - If either check returns `true`, the whole method `canBeIncreasing` returns `true`, indicating the given array can be made strictly increasing by removing one element. If both checks return `false`, the method returns `false`.

In terms of algorithms and patterns, this approach employs a `greedy` strategy, testing if the removal of just one element at the point of violation can make the entire array strictly increasing. No advanced data structures are used, just elementary array and control flow manipulation.

Example Walkthrough

Let's take an example array `nums = [1, 3, 2, 4]` to illustrate the solution approach.

1. We start by iterating through the array from the second element. We compare each element with the one before it.
2. In the first iteration, we have `nums[0] = 1` and `nums[1] = 3`. Since `1 < 3`, the array is strictly increasing so far, and no action is needed.
3. In the second iteration, we see `nums[1] = 3` and `nums[2] = 2`. `3` is not less than `2`, which violates our strictly increasing condition. This is our potential problem area.
4. We now have two scenarios to check:
   - Remove the previous element and check if the new array (ignoring `nums[1]`) is strictly increasing ([`1, 2, 4`]).
   - Remove the current element and check if the new array (ignoring `nums[2]`) is strictly increasing ([`1, 3, 4`]).
5. We call our helper function `check(nums, i)` for both scenarios:
   - `check(nums, 1)` would ignore `nums[1] = 3`, resulting in [`1, 2, 4`]. The sequence is strictly increasing, so this returns `true`.
   - `check(nums, 2)` would ignore `nums[2] = 2`, resulting in [`1, 3, 4`]. This sequence is also strictly increasing, so this would also return `true`.
6. Since removing `nums[1]` (which is `3`) results in a strictly increasing array, we don't need to check further. We can return `true`.

The implementation would look something like this in Python:

```python
class Solution:
    def canBeIncreasing(self, nums):
        def check(nums, i):
            prev = float('-inf')
            for k, num in enumerate(nums):
                if k == i:
                    continue
                if prev >= num:
                    return False
                prev = num
            return True

        for i in range(1, len(nums)):
            if nums[i-1] >= nums[i]:
                return check(nums, i-1) or check(nums, i)

        return True

# Example Usage
sol = Solution()
result = sol.canBeIncreasing([1, 3, 2, 4])
print(result)  # Output should be True
```

In this example, our array `nums = [1, 3, 2, 4]` can indeed be made strictly increasing by removing the element 3 (at index 1), and our function would correctly return `true`.

Python Solution

```python
from typing import List

class Solution:
    def canBeIncreasing(self, nums: List[int]) -> bool:
        # Helper function to check if the sequence is strictly increasing
        # by skipping the element at index skip_index
        def is_strictly_increasing(nums, skip_index):
            prev_value = float('-inf')
            for index, num in enumerate(nums):
                # Skip the element at skip_index
                if index == skip_index:
                    continue
                # If current element is not greater than the previous one, sequence is not increasing
                if prev_value >= nums[index]:
                    return False
                prev_value = nums[index]
            return True

        # Initialize variables
        current_index = 1
        sequence_length = len(nums)

        # Find the first instance where the sequence is not increasing
        while current_index < sequence_length and nums[current_index - 1] < nums[current_index]:
            current_index += 1

        # Check if the sequence can be made strictly increasing
        # by removing the element at the index just before or at the point of discrepancy
        return (is_strictly_increasing(nums, current_index - 1) or
                is_strictly_increasing(nums, current_index))

# Example usage:
# sol = Solution()
# result = sol.canBeIncreasing([1, 2, 10, 5, 7])
# print(result)  # Output: True, since removing 10 makes the sequence strictly increasing

```

Java Solution

```java
class Solution {
    // Function to check if removing one element from the array can make it strictly increasing
    public boolean canBeIncreasing(int[] nums) {
        int currentIndex = 1;
        int arrayLength = nums.length;
        // Iterate over the array to find the breaking point where the array ceases to be strictly increasing
        for (; currentIndex < arrayLength && nums[currentIndex - 1] < nums[currentIndex]; ++currentIndex);

        // Check if it's possible to make the array strictly increasing by removing the element at
        // either the breaking point or the one before it
        return isStrictlyIncreasingAfterRemovingIndex(nums, currentIndex - 1) ||
               isStrictlyIncreasingAfterRemovingIndex(nums, currentIndex);
    }

    // Helper function to check if the array is strictly increasing after removing the element at index i
    private boolean isStrictlyIncreasingAfterRemovingIndex(int[] nums, int indexToRemove) {
        int prevValue = Integer.MIN_VALUE;
        // Iterate over the array
        for (int currentIndex = 0; currentIndex < nums.length; ++currentIndex) {
            // Skip the element at the removal index
            if (currentIndex == indexToRemove) {
                continue;
            }
            // Check if the previous value is not less than the current, array can't be made strictly increasing
            if (prevValue >= nums[currentIndex]) {
                return false;
            }
            // Update previous value
            prevValue = nums[currentIndex];
        }
        return true; // Array can be made strictly increasing after removing the element at indexToRemove
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if it's possible to have a strictly increasing sequence
    // by removing at most one element from the given vector.
    bool canBeIncreasing(vector<int>& nums) {
        // Find the first index where the sequence stops increasing
        int n = nums.size(); // Storing the size of nums
        // Find the first instance where the current element is not greater than the previous one.
        for (i = 1; n > 1 && nums[i - 1] < nums[i]; ++i)
            ; // The loop condition itself ensures increment, empty body

        // Check the sequences by excluding the element at (i - 1) or i
        return isIncreasingSequence(nums, i - 1) || isIncreasingSequence(nums, i);
    }

private:
    // Helper function to determine whether the sequence is strictly increasing
    // if we virtually remove the element at index 'exclusionIndex'.
    bool isIncreasingSequence(vector<int>& nums, int exclusionIndex) {
        int prevVal = INT_MIN; // Initialize to the smallest integer case
        for (int currIndex = 0; currIndex < nums.size(); ++currIndex) {
            if (currIndex == exclusionIndex) continue; // Skip the exclusion index

            // If the current element is not greater than the previous one, it's not strictly increasing.
            if (prevVal >= nums[currIndex]) return false;
            prevVal = nums[currIndex]; // Update the previous value
        }
        return true; // If all checks passed, the sequence is strictly increasing
    }
};
```

Typescript Solution

```typescript
function canBeIncreasing(nums: number[]): boolean {
    // Helper function to check if the array can be strictly increasing
    // by potentially removing the element at position p
    const isStrictlyIncreasingWithRemoval = (positionToRemove: number): boolean => {
        let previousValue: number | undefined = undefined; // Holds the last valid value
        for (let index = 0; index < nums.length; index++) {
            // Skips the element at the removal position
            if (positionToRemove !== index) {
                // Checks if the current element breaks the strictly increasing order
                if (previousValue !== undefined && previousValue >= nums[index]) {
                    return false;
                }
                // Updates the previous value to the current one
                previousValue = nums[index];
            }
        }
        return true;
    };

    // Iterate through the input array to find the break in the strictly increasing sequence
    for (let i = 0; i < nums.length - 1; i++) {
        // Check if the current element is not less than the previous one
        if (i > 0 && nums[i] <= nums[i - 1]) {
            // Attempt to remove either the previous element or the current one
            // and make the sequence strictly increasing
            return isStrictlyIncreasingWithRemoval(i - 1) || isStrictlyIncreasingWithRemoval(i);
        }
    }

    // If no breaks are found, the sequence is already strictly increasing
    return true;
}
```

Time and Space Complexity

The given code aims to determine if a strictly increasing sequence can be made by removing at most one element from the array `nums`.

Time Complexity

The `check()` function is called at most twice, regardless of the input size. It iterates through the `nums` array up to `n` times, where `n` is the length of the array, potentially skipping one element. If we consider the length of the array as `n`, the time complexity of the `check()` is `O(n)` because it involves a single loop through all the elements.

Since `check()` is called at most twice, the time complexity of the entire `canBeIncreasing()` method is `O(n) + O(n)` which simplifies to `O(n)`.

Space Complexity

The space complexity refers to the amount of extra space or temporary storage that an algorithm uses.

In the case of the provided Python code:

- The `check()` function uses a constant amount of additional space (only the `prev` variable is used).
- No additional arrays or data structures are created that depend on the input size `n`.

Therefore, the space complexity of the code is `O(1)`, indicating constant space usage independent of the input size.