674. Longest Continuous Increasing Subsequence

Easy Array Leetcode Link

Problem Description

The task is to find the length of the longest strictly increasing contiguous subsequence within an array of integers, `nums`. A subsequence is considered **continuous increasing** if each element is strictly greater than the preceding one with no interruptions. In more concrete terms, given indices `l` and `r` where `l < r`, the elements at `[nums[l], nums[l + 1], ..., nums[r - 1], nums[r]]` must satisfy `nums[i] < nums[i + 1]` for all `l <= i < r`. The aim is to determine the maximum length of such a subsequence within the given array.

Intuition

To find this maximum length of a continuous increasing subsequence, we iterate through the array, keeping track of the length of the current increasing subsequence (`f`) and the longest increasing subsequence found so far (`res`).

As we move through the array, we compare each element with its predecessor. If the current element is greater than the previous one, it can extend an increasing subsequence; thus, we increment the length of the current increasing subsequence (`f`). If the element does not increase compared to the previous one, it signifies the end of the current increasing subsequence, and we reset the current length (`f`) to 1, starting a new subsequence from this element.

After each step, we need to check if the last computed increasing subsequence length (`f`) is greater than the current maximum length we've found (`res`). If it is, we update `res` with the new maximum length. This process continues until we go through all the array elements. By the end, `res` will hold the length of the longest continuous increasing subsequence in the array.

Solution Approach

The solution uses a simple linear scan of the array, which is an efficient algorithmic pattern suited for this problem. No additional data structures are used, leveraging the original array to find the solution, which grants an `O(1)` space complexity.

Here's how the algorithm works in detail:

1. **Initialization**: Two variables are initialized: `res` and `f`, both with the value `1`. `res` will store the maximum length of a continuous increasing subsequence found so far, and `f` tracks the length of the current increasing subsequence as we iterate through the array.
2. **Iteration**: We then iterate through the array starting from the second element (at index `1`) all the way to the end.
3. **Subsequence Extension**: For every element `nums[i]`, we compare it with the previous element `nums[i - 1]`. If `nums[i]` is greater than `nums[i - 1]`, the current subsequence is increasing, and so we increment `f` by `1`. In essence, the operation is `f = 1 + (f if nums[i - 1] < nums[i] else 0)`, which can be read as "set `f` to `1` plus (continue adding to `f` if the subsequence is increasing, otherwise reset `f` to `1`)".
4. **Update Maximum Length**: After evaluating whether the subsequence can be extended or needs to be restarted, we next update `res` to be the maximum of its current value or `f`. The expression `res = max(res, f)` ensures that `res` always contains the length of the longest continuous increasing subsequence found at any point in the scan.
5. **Result**: After the iteration completes, the value of `res` is the final answer and is returned. This represents the longest length of a continuous increasing subsequence in the array `nums`.

No complex data structures are needed because we only track the length of the subsequences, not the subsequences themselves. The core pattern used here is a single-pass iteration with constant-time checks and updates, leading to an `O(n)` time complexity, where `n` is the number of elements in the array.

Example Walkthrough

Let's take an example array `nums` to illustrate the solution approach: `[2, 6, 4, 7, 8]`.

1. **Initialization**: We start by initializing `res` and `f` to `1`. This is because the minimum length for an increasing subsequence, by default, is `1` (a single element).

   Currently `res = 1`, `f = 1`.

2. **Iteration**: We start iterating from the second element:
   - At index `1`: `nums[1]` is `6`, `nums[0]` is `2`. Since `6 > 2`, the subsequence is increasing. We increment `f`: `f = f + 1 => 2`.
   - Now, we update `res` to be the maximum of `res` and `f`. Since `f` is `2` and `res` is `1`, `res` becomes `2`.

   Current status: `res = 2`, `f = 2`.

3. **Subsequence Extension**:
   - At index `2`: `nums[2]` is `4`, `nums[1]` is `6`. Since `4` is not greater than `6`, we reset `f` to `1`.
   - `res` remains unchanged because it is still holding the maximum found so far which is `2`.

   Current status: `res = 2`, `f = 1`.

4. **Continuing the Iteration**:
   - At index `3`: `nums[3]` is `7`, `nums[2]` is `4`. Since `7 > 4`, we consider this a continuation of an increasing subsequence and increment `f`: `f = 1 + 1 => 2`.
   - Update `res`: It remains `2` since `f` is not greater than `res`.

   Current status: `res = 2`, `f = 2`.

5. **Final Update**:
   - At index `4`: `nums[4]` is `8`, `nums[3]` is `7`. The increasing pattern continues; thus, we increment `f`: `f = 2 + 1 => 3`.
   - Update `res`: `res` becomes `3` because `f` is now greater than `res`.

   Current status (final): `res = 3`, `f = 3`.

6. **Result**: After completing the iteration, we have found that the length of the longest continuous increasing subsequence in `nums` is `3` ([4, 7, 8]), and we return this value.

This walkthrough has shown that the algorithm successfully identifies and tracks the lengths of increasing subsequences and maintains the length of the longest one found as it progresses through the array. With the time complexity of `O(n)` and constant space usage, it is an efficient method for solving this problem.

Python Solution

```python
from typing import List

class Solution:
    def findLengthOfLCIS(self, nums: List[int]) -> int:
        # Initialize the length of the array
        array_length = len(nums)

        # Initialize the result and the current length of longest consecutive increasing subsequence (LCIS)
        result = current_length = 1

        # Loop through the array starting from the second element
        for i in range(1, array_length):
            # If the current number is greater than the previous one, increment current_length
            if nums[i - 1] < nums[i]:
                current_length += 1
            else:
                # Reset current_length if the sequence is not increasing
                current_length = 1

            # Update result with the maximum length found so far
            result = max(result, current_length)

        # Return the length of the longest consecutive increasing subsequence
        return result
```

Java Solution

```java
class Solution {
    public int findLengthOfLCIS(int[] nums) {
        int maxLength = 1; // Initialize maxLength to 1 since the minimal length of subsequence is 1
        int currentLength = 1; // Start with currentLength of 1, this will track the length of the current subsequence

        // Loop through the array starting from the second element
        for (int i = 1; i < nums.length; ++i) {
            // If the current number is greater than the previous one, increase currentLength
            if (nums[i - 1] < nums[i]) {
                currentLength++;
            } else {
                currentLength = 1; // Reset currentLength to 1 if the sequence breaks
            }

            // Update maxLength if we found a longer subsequence
            maxLength = Math.max(maxLength, currentLength);
        }

        return maxLength; // Return the result which is the length of longest continuous increasing subsequence
    }
}
```

C++ Solution

```cpp
#include<vector>
using namespace std;

class Solution {
public:
    // This method finds the length of the longest contiguous increasing subsequence in the vector.
    int findLengthOfLCIS(vector<int>& nums) {
        if (nums.empty()) return 0; // If the vector is empty, return 0 because there's no subsequence.

        int maxLength = 1; // Initialize maxLength to 1 since the minimum length is 1 if the vector is not empty.
        int currentLength = 1; // This will keep track of the current increasing subsequence length.

        // Loop through the vector starting from the second element.
        for (int i = 1; i < nums.size(); ++i) {
            // If the current element is greater than the previous one, increment the currentLength.
            if (nums[i - 1] < nums[i]) {
                currentLength++;
            } else {
                // Otherwise, reset currentLength to 1 because the sequence has been broken.
                currentLength = 1;
            }

            // Update the maxLength if we found a longer subsequence.
            maxLength = max(maxLength, currentLength);
        }

        // Return the length of the longest contiguous increasing subsequence.
        return maxLength;
    }
};
```

Typescript Solution

```typescript
function findLengthOfLCIS(nums: number[]): number {
    // The length of the input array.
    const lengthOfNums = nums.length;
    // Maximum length of the longest continuous increasing subsequence found so far.
    let maxLength = 1;
    // Starting index of the current subsequence under consideration.
    let startIndex = 0;

    // Loop through the array starting from index 1 to compare with previous elements.
    for (let currentIndex = 1; currentIndex < lengthOfNums; currentIndex++) {
        // If the current element is not larger than the previous,
        // handle the end of the current increasing subsequence.
        if (nums[currentIndex - 1] >= nums[currentIndex]) {
            // Update the maxLength with the length of the just-ended subsequence if it's longer.
            maxLength = Math.max(maxLength, currentIndex - startIndex);
            // Update the startIndex to the current index as the start of a new subsequence.
            startIndex = currentIndex;
        }
    }
    // After the loop, compare the final subsequence with the current max length.
    // This handles the case when the longest subsequence reaches the end of the array.
    return Math.max(maxLength, lengthOfNums - startIndex);
}
```

Time and Space Complexity

The code provided calculates the length of the longest continuous increasing subsequence (LCIS) in an array of integers.

Time Complexity

To determine the time complexity, we analyze the number of operations that are performed in relation to the size of the input array `nums`.

The function iterates once over the array, starting from the second element, and performs a constant amount of work for each element by checking if the current element is greater than the previous element and updating the `f` and `res` variables accordingly.

Since there is only one loop over `n` elements of the array, and within each iteration, the operations are performed in constant time, the time complexity of the function is `O(n)`, where `n` is the length of the input array `nums`.

Space Complexity

To determine the space complexity, we analyze the amount of additional memory that the code uses in relation to the size of the input.

The function uses a fixed number of variables: `n`, `res`, and `f`. No additional data structures that grow with input size are used. This means that the space used does not depend on the size of the input array, but is instead constant.

As a result, the space complexity of the function is `O(1)`, indicating that it uses a constant amount of memory regardless of the input size.