673. Number of Longest Increasing Subsequence

Medium Binary Indexed Tree Segment Tree Array Dynamic Programming

Problem Description

The problem we're given involves an integer array called `nums`. Our goal is to figure out how many subsequences of this array have the greatest length and are strictly increasing. A subsequence is defined as a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Importantly, the condition here is that our subsequence must be strictly increasing, meaning that each element must be greater than the one before it.

Intuition

To address the problem, we consider a dynamic programming approach that allows us to build up the solution by considering the number of increasing subsequences ending at each element.

We use two arrays, `dp` and `cnt`. The `dp` array will keep track of the length of the longest increasing subsequence that ends with `nums[i]` for each element `i`. The `cnt` array, conversely, records the total number of such subsequences of maximum length ending with `nums[i]`.

We need to iterate over each number in the `nums` array and, for each element, iterate over all previous elements to check if a longer increasing subsequence can be created. When we find such an element `nums[j]` that is less than `nums[i]` (ensuring an increase), we have two cases:

1. If the subsequence ending at `nums[j]` can be extended by `nums[i]` to yield a longer subsequence, we update `dp[i]` and set `cnt[i]` to be the same as `cnt[j]` since we have identified a new maximum length.
2. If the extended subsequence has the same length as the current maximum subsequence ending at `nums[i]`, we increase `cnt[i]` by `cnt[j]` because we have found an additional subsequence of this maximum length.

As we move through the array, we maintain variables to keep track of the longest subsequence found so far (`maxLen`) and the number of such subsequences (`ans`). If a new maximum length is found, `maxLen` is updated and `ans` is set to the number of such subsequences up to `nums[i]`. If we find additional sequences of the same maximum length, we add to `ans` accordingly.

At the end of the iterations, `ans` represents the total number of longest increasing subsequences present in the whole array `nums`.

Solution Approach

The solution uses a dynamic programming approach, which is a method for solving complex problems by breaking them down into simpler subproblems. It stores the results of these subproblems to avoid redundant work.

Here is a step-by-step breakdown of the approach:

1. Initialize two lists, `dp` and `cnt`, both of size `n`, where `n` is the length of the input array `nums`. Each element of `dp` starts with a value of 1, because the minimum length of an increasing subsequence is 1 (the element itself). Similarly, `cnt` starts with 1s because there's initially at least one way to have a subsequence of length 1 (again, the element itself).
2. Iterate over the array with two nested loops. The outer loop goes from `i = 0` to `i = n - 1`, iterating through the elements of `nums`. The inner loop goes from `j = 0` to `j < i`, considering elements prior to `i`.
3. Within the inner loop, compare the elements at indices `j` and `i`. If `nums[i]` is greater than `nums[j]`, it means `nums[i]` can potentially extend an increasing subsequence ending at `nums[j]`.
4. Check if the increasing subsequence ending at `j` plus 1 is greater than the current subsequence length at `i` (`dp[j] + 1 > dp[i]`). If it is, this means we've found a longer increasing subsequence ending at `i`, so update `dp[i]` to `dp[j] + 1`, and set `cnt[i]` to `cnt[j]` because the number of ways to achieve this longer subsequence at `i` is the same as the number at `j`.
5. If `dp[j] + 1` equals `dp[i]`, it indicates another subsequence of the same maximum length ending at `i`, so increment `cnt[i]` by `cnt[j]`.
6. Keep track of the overall longest sequence length found (`maxLen`) and the number of such sequences (`ans`). After processing each `i`, compare `dp[i]` with `maxLen`. If `dp[i]` is greater than `maxLen`, update `maxLen` to `dp[i]` and set `ans` to `cnt[i]`. If `dp[i]` equals `maxLen`, add `cnt[i]` to `ans`.
7. Continue this process until all elements have been considered. At the end, the variable `ans` will contain the number of longest increasing subsequences in `nums`.

By using dynamic programming, this solution effectively builds up the increasing subsequences incrementally and keeps track of their counts without having to enumerate each subsequence explicitly, which would be infeasible due to the exponential number of possible subsequences.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose the given integer array `nums` is `[3, 1, 2, 3, 1, 4]`.

1. We initialize `dp` and `cnt` arrays as `[1, 1, 1, 1, 1, 1]` since each element by itself is a valid subsequence of length 1 and there is at least one way to make a subsequence of one element.
2. Start iterating with `i` from index 0 to the end of the array, and for each `i`, we iterate with `j` from 0 to `i-1`.
3. When `i = 2` (element is 2), we find that `nums[2] > nums[0]` (2 > 3). However, as 2 is not greater than 3, no updates are done. Then we compare `nums[2]` with `nums[1]` (2 > 1), this is true, and since `dp[1] + 1 > dp[2]`, we update `dp[2]` to `dp[1] + 1 = 2`, also updating `cnt[2]` to `cnt[1]`.
4. Moving to `i = 3` (element is 3), we compare with all previous elements one by one. For `j = 0`, `nums[3]` (3) is not greater than `nums[0]` (3). For `j = 1`, `nums[3]` (3) is greater than `nums[1]` (1), and so `dp[1] + 1 > dp[3]` so we update `dp[3]` and `cnt[3]` to 2. For `j = 2`, `nums[3]` (3) is greater than `nums[2]` (2), and `dp[2] + 1 = 3` which is greater than the current `dp[3]` (2). So we update `dp[3]` to 3, and `cnt[3]` to `cnt[2]`.
5. Continue this process for `i = 4` and `i = 5`. When we reach `i = 5` (element is 4) and `j = 3`, since `nums[5]` (4) is greater than `nums[3]` (3) and `dp[3] + 1 > dp[5]` (4 > 1), we update `dp[5]` to 4, and `cnt[5]` to `cnt[3]`.
6. We keep track of the `maxLen` and `ans` throughout the process. Initially, `maxLen` is 1 and `ans` is 6 (because we have six elements, each a subsequence of length 1). After updating all `dp[i]` and `cnt[i]`, we find `maxLen` to be 4 (subsequences ending at index 5) and `ans` will be updated to the count of the longest increasing subsequences ending with `nums[5]`, which in this case is 1.

At the end of this process, the `dp` array is `[1, 1, 2, 3, 1, 4]` and the `cnt` array is `[1, 1, 1, 1, 1, 1]`. We conclude that the length of the longest increasing subsequence is 4 and the number of such subsequences is 1 (ending with the element 4 in the original array).

This walk-through illuminates the step-by-step application of our dynamic programming solution to find the total number of the longest increasing subsequences present in the array `nums`.

Python Solution

```python
class Solution:
    def findNumberOfLIS(self, nums: List[int]) -> int:
        # Initialize variables
        max_length = 0  # The length of the longest increasing subsequence
        answer = 0  # The number of longest increasing subsequences
        num_elements = len(nums)  # Number of elements in the input list
        dp = [1] * num_elements  # dp[i] will store the length of the LIS ending at nums[i]
        count = [1] * num_elements  # count[i] will store the number of LIS's ending at nums[i]

        # Iterate through the list to populate dp and count arrays
        for i in range(num_elements):
            for j in range(i):
                # Check if the current element is greater than the previous ones to form an increasing sequence
                if nums[i] > nums[j]:
                    # If we found a longer increasing subsequence ending at i
                    if dp[j] + 1 > dp[i]:
                        dp[i] = dp[j] + 1
                        count[i] = count[j]
                    # If we found another LIS of the same length as the longest found ending at i
                    elif dp[j] + 1 == dp[i]:
                        count[i] += count[j]

            # Update the global maximum length and the number of such sequences
            if dp[i] > max_length:
                max_length = dp[i]
                answer = count[i]
            elif dp[i] == max_length:
                answer += count[i]

        # Return the total number of longest increasing subsequences
        return answer
```

Java Solution

```java
class Solution {
    public int findNumberOfLIS(int[] nums) {
        int maxLength = 0; // Store the length of the longest increasing subsequence
        int numberOfMaxLIS = 0; // Store the count of longest increasing subsequences
        int n = nums.length; // The length of the input array

        // Arrays to store the length of the longest subsequence up to each element
        int[] lengths = new int[n]; // dp[i] will be the length for nums[i]
        int[] counts = new int[n]; // cnt[i] will be the number of LIS for nums[i]

        for (int i = 0; i < n; i++) {
            lengths[i] = 1; // By default, the longest subsequence at each element is 1 (the element itself)
            counts[i] = 1; // By default, the count of subsequences at each element is 1

            // Check all elements before the i-th element
            for (int j = 0; j < i; j++) {
                // If the current element can extend the increasing subsequence
                if (nums[j] < nums[i]) {
                    // If a longer subsequence is found
                    if (lengths[j] + 1 > lengths[i]) {
                        lengths[i] = lengths[j] + 1; // Update the length for nums[i]
                        counts[i] = counts[j]; // And update the count for nums[i]
                    } else if (lengths[j] + 1 == lengths[i]) {
                        counts[i] += counts[j]; // If same length, add the count of subsequences from nums[j]
                    }
                }
            }

            // If a new maximum length of subsequence is found
            if (lengths[i] > maxLength) {
                maxLength = lengths[i]; // Update the maxLength with the new maximum
                numberOfMaxLIS = counts[i]; // Reset the count of LIS
            } else if (lengths[i] == maxLength) {
                // If same length subsequences are found, add the count to the total
                numberOfMaxLIS += counts[i];
            }
        }

        // Return the total count of longest increasing subsequences
        return numberOfMaxLIS;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the number of longest increasing subsequences
    int findNumberOfLIS(vector<int>& nums) {
        int n = nums.size(); // Size of the nums array
        if (n <= 1) return n; // If array is empty or has one element, return n

        int maxLength = 0; // Variable to store length of longest increasing subsequence
        int numberOfLIS = 0; // Variable to store number of longest increasing subsequences
        vector<int> lengths(n, 1); // dp array - lengths[i] will store the length of LIS ending with nums[i]
        vector<int> counts(n, 1); // Array to store the count of LIS of the same length ending with nums[i]

        // Iterate over the array
        for (int i = 0; i < n; ++i) {
            // Check all previous numbers
            for (int j = 0; j < i; ++j) {
                // If nums[i] can be appended to the LIS ending with nums[j]
                if (nums[i] > nums[j]) {
                    // If a longer subsequence ending with nums[i] is found
                    if (lengths[j] + 1 > lengths[i]) {
                        lengths[i] = lengths[j] + 1; // Update the length of LIS for nums[i]
                        counts[i] = counts[j]; // Reset the count for nums[i] to the counts of nums[j]
                    } else if (lengths[j] + 1 == lengths[i]) {
                        // If another LIS of the same length ending with nums[i] is found
                        counts[i] += counts[j]; // Increment the count for nums[i]
                    }
                }
            }

            // Update global max length and count of the LIS
            if (lengths[i] > maxLength) {
                maxLength = lengths[i]; // Update the maximum length
                numberOfLIS = counts[i]; // Update the number of LIS
            } else if (lengths[i] == maxLength) {
                numberOfLIS += counts[i]; // If same length, add the count for nums[i] to global count
            }
        }

        return numberOfLIS; // Return the number of longest increasing subsequences
    }
};
```

Typescript Solution

```typescript
// Define types for better code readability
type NumberArray = number[];

// Function to find the number of longest increasing subsequences
function findNumberOfLIS(nums: NumberArray): number {
    let n: number = nums.length; // Size of the nums array
    if (n <= 1) return n; // If array is empty or has one element, return n

    let maxLength: number = 0; // Variable to store length of longest increasing subsequence
    let numberOfLIS: number = 0; // Variable to store number of longest increasing subsequences
    let lengths: NumberArray = new Array(n).fill(1); // Array to store lengths of LIS up to nums[i]
    let counts: NumberArray = new Array(n).fill(1); // Array to store counts of LIS of same length at nums[i]

    // Iterate over the array to build up lengths and counts
    for (let i = 0; i < n; ++i) {
        // Check all elements before i
        for (let j = 0; j < i; ++j) {
            // If current element nums[i] can extend the LIS ending at nums[j]
            if (nums[i] > nums[j]) {
                // Found a longer subsequence ending at nums[i]
                if (lengths[j] + 1 > lengths[i]) {
                    lengths[i] = lengths[j] + 1; // Update the LIS length at i
                    counts[i] = counts[j]; // Reset counts at i to those at j
                } else if (lengths[j] + 1 === lengths[i]) {
                    // Found another LIS of the same length ending at i
                    counts[i] += counts[j]; // Increment counts at i
                }
            }
        }
        // Update maxLength and numberOfLIS if needed
        if (lengths[i] > maxLength) {
            maxLength = lengths[i]; // Update max length with the new longest found
            numberOfLIS = counts[i]; // Set the number of LIS to the count at i
        } else if (lengths[i] === maxLength) {
            numberOfLIS += counts[i]; // Another LIS found - increase the number of LIS
        }
    }

    return numberOfLIS; // Return the number of longest increasing subsequences found
}
```

Time and Space Complexity

The given Python function `findNumberOfLIS` calculates the number of Longest Increasing Subsequences (LIS) in an array of integers. Analyzing the time complexity involves examining the nested loop structure, where the outer loop runs `n` times (once for each element in the input list `nums`) and the inner loop runs at most `i` times for the `i`th iteration of the outer loop. Consequently, in the worst case, the inner loop executes `1 + 2 + 3 + ... + (n - 1)` times, which can be expressed as the sum of the first `n-1` natural numbers, resulting in a time complexity of $O(n^2)$.

The space complexity of the function is determined by the additional memory allocated for the dynamic programming arrays `dp` and `cnt`, each of size `n`. Therefore, the space complexity of the function is $O(n)$, which is linear with respect to the size of the input list.