903. Valid Permutations for DI Sequence

Hard | String | Dynamic Programming | Prefix Sum | Leetcode Link

Problem Description

Given a string `s` of length `n`, which consists of characters 'D' and 'I' representing decreasing and increasing respectively, we are tasked with generating a permutation `perm` of integers from `0` to `n`. This permutation must satisfy the condition where 'D' at index `i` means `perm[i]` should be greater than `perm[i + 1]`, and 'I' means `perm[i]` should be less than `perm[i + 1]`. The goal is to find the total number of such valid permutations and return this number modulo `10^9 + 7` since the result could be very large.

Intuition

To solve this problem, we need to leverage dynamic programming (DP) because the straightforward backtracking approach would be too slow due to its exponential time complexity.

The intuition behind this DP approach is to keep track of the number of valid sequences at each step while building the permutation from left to right according to the given 'D' and 'I' constraints. We use an auxiliary array `f` to represent the number of valid permutations ending with each possible number at each step.

We iterate over the input string `s`, and for each character, we update the DP array based on whether it's 'D' or 'I'. For 'D', we want to accumulate the counts in reverse since we're adding a smaller number before a larger one. For 'I', we accumulate counts forward since we're adding a larger number after the smaller one. This leads us to construct a new array `g` at each step, representing the new state of the DP.

At each step, `g` gets the cumulative sum of values from `f` depending on the 'D' or 'I' condition. By the end of the `s` traversal, the `f` array holds the counts of valid permutations ending with each number. The total number of distinct valid permutations would be the sum of all values in this DP array `f`. We return this sum modulo `10^9 + 7`.

Solution Approach

The given Python solution employs dynamic programming. Let's elaborate on its implementation step by step:

- `mod = 10**9 + 7` ensures that we take the remainder after division with `10^9 + 7`, satisfying the problem's requirement to return results under this modulo to handle large numbers.
- The list `f` is initialized with `1` followed by `n` zeros: `f = [1] + [0] * n`. The `1` represents the number of valid permutations when the permutation length is 0 (which is 1 since there's only an empty permutation), and the zeros act as placeholders for the future steps.
- The `for` loop over the enumerated string `s` is the core of the dynamic programming:
  - `pre` is initialized to zero. It serves as a prefix sum that gets updated while iterating through the positions where a number could be placed.
  - A temporary list `g` is initialized with zeros. It'll be populated with the number of valid permutations ending at each index for the current length of the permutation sequence.
  - Inside this loop, depending on whether the current character `c` is 'D' or 'I', we iterate through the list `f` to update the prefix sums.
    - If `c` is "D": We iterate backward because we want to place a number that's smaller than the preceding one. As we iterate, we update `pre` by adding `f[j]` modulo `mod` and assign `pre` to `g[j]`.
    - If `c` is "I": We iterate forward because we want to place a number larger than the previous one. Here, `g[j]` is assigned the value of `pre`, and then `pre` is updated to its value plus `f[j]`, also modulo `mod`.
- After processing either 'D' or 'I' for all positions, we replace `f` with the new state `g`. This assignment progresses our dynamic programming to the next step, where `f` now represents the state of the DP for sequences of length `i + 1`.
- Once we finish going through the string `s`, `f` will contain the number of ways to arrange permutations ending with each possible number for a sequence of length `n`. This is because each iteration effectively builds upon the previous length, considering whether we're adding a number in a decremented ('D') or incremented ('I') fashion.
- The final result is the sum of all counts in `f` modulo `mod`, which gives us the total count of valid permutations that can be formed according to the input pattern string `s`.

The use of a rolling DP array `f` that gets updated at each step with `g` and the accumulation of counts in forward or backward fashion depending on 'I' or 'D' characters are the lynchpins of this solution. This approach optimizes computing only the necessary states at each step without having to look at all permutations explicitly, which would be prohibitively expensive for large `n`.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose the string `s` is `"DID"`. The length of `s` is `3`, so our permutation `perm` needs to be comprised of integers from `0` to `3`.

1. **Initial State:** We initialize `f` with `1` plus `3` zeros: `f = [1, 0, 0, 0]`. The `1` represents the singular empty permutation, and zeros are placeholders to be filled.
2. **First Character ('D'):** The first character is 'D', so we want `perm[0] > perm[1]`. We iterate backward. Say, initially, `f = [1, 0, 0, 0]`, and we are filling `g`.
   - `pre = 0` (prefix sum)
   - Start from the end of `f`, `pre = pre + f[3] % mod = 0` and `g[3] = pre = 0`
   - `pre = pre + f[2] % mod = 0` and `g[2] = pre = 0`
   - `pre = pre + f[1] % mod = 0` and `g[1] = pre = 0`
   - `pre = pre + f[0] % mod = 1` and `g[0] = pre = 1` After this, `g = [1, 0, 0, 0]` and we update `f` to `g`.
3. **Second Character ('I'):** The second character is 'I', so we want `perm[1] < perm[2]`. We iterate forward.
   - `pre = 0` (reset prefix sum)
   - `g[0] = pre = 0`
   - `pre = pre + f[0] % mod = 1` and `g[1] = pre = 1`
   - `pre = pre + f[1] % mod = 1` and `g[2] = pre = 1`
   - `pre = pre + f[2] % mod = 1` and `g[3] = pre = 1` After this step, `g = [0, 1, 1, 1]` and we update `f` to `g`.
4. **Third Character ('D'):** The third character is 'D', so `perm[2] > perm[3]`. We iterate backward again.
   - `pre = 0`
   - `pre = pre + f[3] % mod = 1` and `g[3] = pre = 1`
   - `pre = pre + f[2] % mod = 2` and `g[2] = pre = 2`
   - `pre = pre + f[1] % mod = 3` and `g[1] = pre = 3`
   - We do not need to continue, as `f[0]` represents a state where the sequence is already longer than what the 'D' at last position can influence. After this, `g = [0, 3, 2, 1]` and `f` is updated to `g`.
5. **Final Result:** We sum up the elements in `f` to find the total number of valid permutations: `0 + 3 + 2 + 1 = 6`. The possible permutation patterns are "3210", "3201", "3102", "2103", "3012", and "2013".
6. **Return Value:** We return this sum modulo `10^9 + 7` for the result. In this case, since `6` is less than `10^9 + 7`, the modulus operation is not necessary, and the result is simply `6`.

The above walkthrough visualizes the dynamic programming method, with `f` representing the state of the permutation counts at each step, updated according to the 'D' and 'I' constraints in the input string `s`.

Python Solution

```python
class Solution:
    def numPermsDISequence(self, sequence: str) -> int:
        # Defining the modulus for the problem, as the permutations
        # can be a large number and we need to take it modulo 10^9 + 7
        modulus = 10**9 + 7

        # Length of the input sequence
        n = len(sequence)

        # Initializing the dynamic programming table with the base case:
        # There is 1 permutation of length 0
        dp = [1] + [0] * n

        # Looping through the characters in the sequence along with their indices
        for i, char in enumerate(sequence, 1):
            # 'pre' is used to store the cumulative sum that helps in updating dp table
            pre = 0
            # Initializing a new list to store the number of permutations for the current state
            new_dp = [0] * (n + 1)

            if char == "D":
                # If the character is 'D', we count the permutations in decreasing order
                for j in range(i, -1, -1):
                    pre = (pre + dp[j]) % modulus
                    new_dp[j] = pre
            else:
                # If the character is 'I', we do the same in increasing order
                for j in range(i + 1):
                    new_dp[j] = pre
                    pre = (pre + dp[j]) % modulus

            # Update the dynamic programming table with new computed values for the next iteration
            dp = new_dp

        # The result is the sum of all permutations possible given the entire DI sequence
        return sum(dp) % modulus

```

Java Solution

```java
class Solution {
    public int numPermsDISequence(String s) {
        final int MODULO = (int) 1e9 + 7; // Define the modulo constant for avoiding overflow.
        int n = s.length(); // Length of the input string.
        int[] dp = new int[n + 1]; // dp array for dynamic programming, initialized for a sequence of length 0.
        dp[0] = 1; // There's one permutation for an empty sequence.

        // Iterate over the sequence.
        for (int i = 1; i <= n; ++i) {
            int cumulativeSum = 0; // To store the cumulative sum for 'D' or 'I' scenarios.
            int[] newDp = new int[n + 1]; // Temporary array to hold new dp values for current iteration.

            // If the character is 'D', we calculate in reverse.
            if (s.charAt(i - 1) == 'D') {
                for (int j = i; j >= 0; --j) {
                    cumulativeSum = (cumulativeSum + dp[j]) % MODULO;
                    newDp[j] = cumulativeSum;
                }
            } else { // Otherwise, we calculate in the forward direction for 'I'.
                for (int j = 0; j <= i; ++j) {
                    newDp[j] = cumulativeSum;
                    cumulativeSum = (cumulativeSum + dp[j]) % MODULO;
                }
            }
            // Assign the newly computed dp values to be used in the next iteration.
            dp = newDp;
        }

        int ans = 0;
        // Sum up all the possible permutations calculated in dp array.
        for (int j = 0; j <= n; ++j) {
            ans = (ans + dp[j]) % MODULO;
        }

        return ans; // Return the total permutations count.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int numPermsDISequence(string s) {
        const int MOD = 1e9 + 7; // Constant to hold the modulus value for large numbers
        int sequenceLength = s.size(); // The length of the sequence
        vector<int> dp(sequenceLength + 1, 0); // Dynamic programming table
        dp[0] = 1; // Base case initialization

        // Iterate over the characters in the input string
        for (int i = 1; i <= sequenceLength; ++i) {
            int accumulated = 0;
            vector<int> nextDP(sequenceLength + 1, 0); // Create a new DP array for the next iteration

            // Check if the current character is 'D' for a decreasing relationship
            if (s[i - 1] == 'D') {
                // Fill in the DP table backwards for 'D'
                for (int j = i; j >= 0; --j) {
                    accumulated = (accumulated + dp[j]) % MOD; // Update accumulated sum
                    nextDP[j] = accumulated; // Update the next DP table
                }
            } else {
                // Else, this is an increasing relationship represented by 'I'
                for (int j = 0; j <= i; ++j) {
                    nextDP[j] = accumulated; // Set the value for 'I'
                    accumulated = (accumulated + dp[j]) % MOD; // Update the accumulated sum
                }
            }
            dp = move(nextDP); // Move the next DP table into the current DP table for the next iteration
        }

        // Sum all possibilities from the last DP table to get the final answer
        int answer = 0;
        for (int j = 0; j <= sequenceLength; ++j) {
            answer = (answer + dp[j]) % MOD;
        }

        return answer; // Return the total number of permutations that match the DI sequence
    }
};

```

Typescript Solution

```typescript
// This function calculates the number of permutations of the sequence of 0...N
// that satisfy the given "DI" (decrease/increase) sequence.
// s: The input "DI" sequence as a string.
// Returns the number of valid permutations modulo 10^9 + 7.
function numPermsDISequence(s: string): number {
    const sequenceLength = s.length;
    let dp: number[] = Array(sequenceLength + 1).fill(0);
    dp[0] = 1; // Base case: one permutation of an empty sequence
    const MOD = 10 ** 9 + 7; // Define the modulo to prevent overflow

    // Iterating over the characters in the sequence
    for (let i = 1; i <= sequenceLength; ++i) {
        let prefixSum = 0; // Initialize prefix sum for the number of permutations
        let nextDp: number[] = Array(sequenceLength + 1).fill(0); // Temporary array to hold new dp values

        // If the current character is 'D', count decreasing sequences
        if (s[i - 1] === 'D') {
            for (let j = i; j >= 0; --j) {
                prefixSum = (prefixSum + dp[j]) % MOD;
                nextDp[j] = prefixSum;
            }
        }
        // If the current character is 'I', count increasing sequences
        else {
            for (let j = 0; j <= i; ++j) {
                nextDp[j] = prefixSum;
                prefixSum = (prefixSum + dp[j]) % MOD;
            }
        }

        // Update the dp array for the next iteration
        dp = nextDp;
    }

    // Sum up all the permutations stored in dp array to get the answer
    let result = 0;
    for (let j = 0; j <= sequenceLength; ++j) {
        result = (result + dp[j]) % MOD;
    }

    // Return the total number of permutations modulo 10^9 + 7
    return result;
}

```

Time and Space Complexity

The given Python code defines a method to count the number of permutations that satisfy a certain "decreasing/increasing" (D/I) sequence requirement, following the rules specified by a given string `s`.

Time Complexity

To analyze the time complexity, let's consider the size of the input string `s`, denoted as `n`. The code includes a main outer loop, which iterates over the input string `s`, and two inner loops, which will both iterate at most `i + 1` times (where `i` ranges from 1 to `n` inclusive).

The main loop runs exactly `n` times: `for i, c in enumerate(s, 1)`. In each iteration, depending on whether the character is a 'D' or not, it performs one of the two inner loops. These inner loops execute a maximum of `i` iterations in their respective contexts (`for j in range(i, -1, -1)` for 'D' and `for j in range(i + 1)` for 'I'), which can be summarized as an arithmetic progression sum from `1` to `n`. Arithmetically summing this progression, we get `n * (n + 1) / 2`.

Consequently, the overall time complexity is `O(n^2)`, since `(n * (n + 1)) / 2` is within the same order of magnitude as `n^2`.

Space Complexity

For space complexity, the script uses a list `f` of size `n + 1` to store the running totals of valid permutations and a temporary list `g` with the same size to calculate the new values for the next iteration.

Since the largest data structure size is proportional to the input size `n`, the space complexity is `O(n)`, which is linear to the input size.