1977. Number of Ways to Separate Numbers

Hard | String | Dynamic Programming | Suffix Array | Leetcode Link

Problem Description

In this problem, we have a string `num` which consists of concatenated positive integers. We're told that the original list of integers was written in a non-decreasing order and none of the integers had leading zeros. The challenge is to figure out how many different lists of positive integers could have been combined to form the initial string `num`.

Since the answer could be very large, we need to return it modulo `10^9 + 7`.

To illustrate with an example, if the input string `num` is "327", there are two possible lists of non-decreasing integers: [3, 27] and [327]. Hence the function would return 2.

Intuition

Approaching this problem requires that we solve it piece by piece. We need to determine under what conditions we can split the string `num` into different integers such that they maintain a non-decreasing order. We can start from the first character and continue appending the following characters until we get a valid number, then process the remainder of the string in a similar manner.

Here's how we might think about the solution:

1. Dynamic Programming (DP): We can consider this problem as one that can be solved using dynamic programming. We try to build up the number of ways to form the string `num` from smaller subproblems based on the length of the last number used.
2. Longest Common Prefix (LCP): To maintain the non-decreasing order of numbers, we need a way to compare the numbers in `num` efficiently. The LCP (Longest Common Prefix) array can be used to store the length of the longest common prefix between two substrings of num. This helps us compare numbers quickly without generating all substrings, which would be inefficient.
3. Construction of DP Table: We construct a 2D DP table with `dp[i][j]` meaning the number of ways to form the first `i` digits of `num` where the last number used has `j` digits.
4. Transition: We consider two cases for the transition. Either the last two numbers used are the same length - in which case we can form a new list of integers by using the same length of the number only if the two numbers are in non-decreasing order. Or, the last number used is shorter than the previous - in which case you can always append it because it won't violate the non-decrease constraint.
5. Base Case: We initiate `dp[0][0]` as 1 because there is one way to form an empty string with an empty list.

This solution iteratively builds upon smaller subproblems and combines their results to find the total number of combinations for larger lengths of the string `num`. At each step, it ensures that numbers are added in a non-decreasing order and without leading zeros.

Solution Approach

The solution utilizes a nested function `cmp` to compare substrings of `num`, and a Dynamic Programming (DP) table to keep track of the number of valid combinations that can be formed up to certain points in the string `num`.

Let's walk through the components of the implementation:

1. **Longest Common Prefix (LCP) Calculation**: Before the main DP logic, the function computes the LCP array using a nested loop. For each pair of indices `i` and `j` in the string `num`, `lcp[i][j]` is calculated to store the length of the longest common prefix between substrings starting from `i` and `j`. This pre-computation is essential for the `cmp` function.
2. **Comparison function (`cmp`)**: The function `cmp(i, j, k)` is used to compare two numbers within the string `num` of length `k` starting from `i` and `j` respectively. This function will indirectly utilize the precomputed LCP values to efficiently determine if the first number starting at index `i` is greater than or equal to the second number starting at index `j`.
3. **Dynamic Programming**: The 2D DP table `dp` is created to keep track of valid combinations. `dp[i][j]` represents the number of ways to partition `num[0:i]` such that the last number used has `j` digits.
4. **DP Initialization**: The `dp` table is initialized such that `dp[0][0]` is 1, since there is exactly one way to create an empty partition.
5. **DP Iteration**: The code iterates through each end index `i` of the substring and considers all possible lengths `j` for the last number. Inside this nested loop, the algorithm decides whether the substring of length `j` ending at `i` can form a valid number (it cannot start with '0') and if it maintains the non-decreasing property compared to the previous number.
6. **DP Transition**: The crucial part of this solution is how the DP values transition from one state to another. For each `i` and `j`, the implementation considers the number of combinations if the last two numbers are of equal length (and the second is greater or equal) or if the second number may length less.
    - If the last two numbers are of equal length and the comparison via `cmp` is true, it takes the value from `dp[i-j][j]` (which represents the count up to the first number's end index and the subsequent number also ending at `i-j` and also has length `j`).
    - Otherwise, it uses the value from `dp[i-j][min(j-1,i-j)]` to represent scenarios wherein the most recent number is of smaller size and doesn't threaten the non-decreasing order.
7. **DP State Updates**: After ascertaining which previous DP state can be transitioned from, `dp[i][j]` is updated with the count of combinations by considering the previous number lengths into account. It sums the count from the previously calculated number of ways and the new count, keeping in mind that we must perform this addition modulo `10^9 + 7`.
8. **Result Extraction**: Lastly, after populating the DP table, the result is the total combinations possible which is `dp[n][n]`, meaning the number of combinations to form the entire string with all possible last number lengths considered.

The solution is a well-crafted example of combining DP with string comparisons optimized by LCP to ensure polynomial time complexity and manage the large answer space creatively with modular arithmetic.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose our input string `num` is "1123". We want to find out in how many different ways we can split this string into a list of non-decreasing positive integers.

Step 1: Longest Common Prefix (LCP) Calculation: We calculate the LCP for every pair of starting positions in the string `num`. For simplicity, let's assume we have a function that does this efficiently.

Step 2: Comparison function (`cmp`): This function will compare two numbers within the string `num` based on their starting indices and length using the LCP information.

Step 3: Dynamic Programming (DP): We initialize a DP table `dp` where `dp[i][j]` will represent the number of ways to split the substring `num[0:i]` such that the last number has length `j`.

Step 4: DP Initialization: We set `dp[0][0]` to 1 because there's one way to form an empty string which is an empty list of integers.

Step 5: DP Iteration:

- For `i = 1`, the substring "1" can form one list, `[1]`.
- For `i = 2`, the substring "11" can form two lists, `[1, 1]` or `[11]`.
- For `i = 3`, the substring "112" can form three lists, `[1, 1, 2]`, `[1, 12]`, or `[11, 2]`.
- For `i = 4`, the substring "1123" can form the following lists: [1, 1, 2, 3], [1, 1, 23], [1, 12, 3], [1, 123], [11, 2, 3], [11, 23]. That's six valid lists.

Step 6: DP Transition:

- We check if we can append a number to the existing one to make sure they are in non-decreasing order.
- For instance, at `i = 4, j = 2`, we compare "23" with "12" (previous number of length 2). Since "23" >= "12", we can consider the count from `dp[2][2]`.

Step 7: DP State Updates: We update the `dp` table by adding the number of ways to the current count, ensuring we consider each possible length of the last number.

Step 8: Result Extraction: We find the total count of combinations to form the string "1123", which will be `dp[4][4] + dp[4][3] + dp[4][2] + ... + dp[4][1]` after iterating through all possible lengths for the last number.

For this example, by walking through the DP table, we find that there are six valid lists that can be formed from "1123". This approach ensures we explore all possibilities iteratively while keeping the problem manageable using the principles of dynamic programming.

Python Solution

```python
class Solution:
    def numberOfCombinations(self, num: str) -> int:
        # Helper function to compare the lexicographical order of two numbers within 'num'
        # representing the start indices and their length 'length'.
        def is_greater_or_equal(start1, start2, length):
            common_prefix_length = longest_common_prefix[start1][start2]
            return common_prefix_length >= length or num[start1 + common_prefix_length] >= num[start2 + common_prefix_length]

        MOD = 10**9 + 7
        num_length = len(num)

        # Pre-compute the longest common prefix (LCP) array.
        longest_common_prefix = [[0] * (num_length + 1) for _ in range(num_length + 1)]
        for i in range(num_length - 1, -1, -1):
            for j in range(num_length - 1, -1, -1):
                if num[i] == num[j]:
                    longest_common_prefix[i][j] = 1 + longest_common_prefix[i + 1][j + 1]

        # Dynamic programming table where dp[i][j] represents the number of combinations
        # of valid integers with length 'j' that end at index 'i - 1'.
        dp = [[0] * (num_length + 1) for _ in range(num_length + 1)]
        dp[0][0] = 1

        # Build up the DP table.
        for i in range(1, num_length + 1):
            for j in range(1, i + 1):
                current_value = 0
                if num[i - j] != '0':  # Skip numbers with leading zero.
                    if i - j - j >= 0 and is_greater_or_equal(i - j, i - j - j, j):
                        # The substring is greater or equal to the previous,
                        # so we can safely append to the previous.
                        current_value = dp[i - j][j]
                    else:
                        # Take the combination counts from the smaller number if present.
                        current_value = dp[i - j][min(j - 1, i - j)]

                # Update the current dp value including the current value with MOD.
                # Accumulate with the previous j-1 combinations.
                dp[i][j] = (dp[i][j - 1] + current_value) % MOD

        # Total number of combinations is the last value in the dp table.
        return dp[num_length][num_length]

```

Java Solution

```java
class Solution {
    private static final int MOD = (int) 1e9 + 7;

    public int numberOfCombinations(String num) {
        int length = num.length();
        // Initialize the longest common prefix (LCP) array.
        int[][] longestCommonPrefix = new int[length + 1][length + 1];

        // Calculate the LCP for each substring pair.
        for (int i = length - 1; i >= 0; --i) {
            for (int j = length - 1; j >= 0; --j) {
                if (num.charAt(i) == num.charAt(j)) {
                    longestCommonPrefix[i][j] = 1 + longestCommonPrefix[i + 1][j + 1];
                }
            }
        }

        // dp[i][j] will hold the number of ways to partition the string ending at i using a number ending at j.
        int[][] dp = new int[length + 1][length + 1];
        dp[0][0] = 1; // Base case: one way to partition an empty string.

        // Build the dp array from the bottom up.
        for (int i = 1; i <= length; ++i) {
            for (int j = 1; j <= i; ++j) {
                int currentVal = 0;

                // We should not start with a leading zero.
                if (num.charAt(i - j) != '0') {
                    // There should be enough characters for the previous number.
                    if (i >= 2 * j) {
                        int commonPrefixLength = longestCommonPrefix[i - j][i - 2 * j];

                        // Check if the current number is greater than or equal to the previous number.
                        if (commonPrefixLength >= j || num.charAt(i - j + commonPrefixLength) >= num.charAt(i - 2 * j + commonPrefi
                            currentVal = dp[i - j][j];
                        }
                    }
                    // If the number is not valid, use the number of ways of the smaller previous number.
                    if (currentVal == 0) {
                        currentVal = dp[i - j][Math.min(j - 1, i - j)];
                    }
                }

                // Sum the ways, and do mod to avoid overflow.
                dp[i][j] = (dp[i][j - 1] + currentVal) % MOD;
            }
        }
        // The answer is the number of ways to form the number sequence using the entire string.
        return dp[length][length];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    const int MOD = 1e9 + 7; // Define modulo value for the results to prevent overflow

    // This function counts the number of non-empty increasing sequences of integers
    // that can be formed by the digits of the given string 'num'
    int numberOfCombinations(string num) {
        int n = num.size(); // Size of the input string
        // 2D vector to store the longest common prefix (LCP) lengths between suffixes of 'num'
        vector<vector<int>> longestCommonPrefix(n + 1, vector<int>(n + 1));

        // Preprocess to fill in the longestCommonPrefix array
        for (int i = n - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                if (num[i] == num[j]) {
                    longestCommonPrefix[i][j] = 1 + longestCommonPrefix[i + 1][j + 1];
                }
            }
        }

        // Comparator function to compare two substrings in 'num' when one is a prefix of the other
        auto canPlace = [&](int start, int prevStart, int len) {
            int commonPrefixLen = longestCommonPrefix[start][prevStart];
            return commonPrefixLen >= len || num[start + commonPrefixLen] >= num[prevStart + commonPrefixLen];
        };

        // 2D vector for dynamic programming (dp), where dp[i][j] represents the number of ways
        // to form an increasing sequence upto index i, using a last number of length j
        vector<vector<int>> dp(n + 1, vector<int>(n + 1));
        dp[0][0] = 1; // Base case: 1 way to form a sequence of length 0

        // Populate the dp array
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= i; ++j) {
                int count = 0;
                // Check if the current number doesn't start with '0' and if it can be placed in the sequence
                if (num[i - j] != '0') {
                    if (i - j - j >= 0 && canPlace(i - j, i - j - j, j)) {
                        count = dp[i - j][j];
                    } else {
                        count = dp[i - j][min(j - 1, i - j)];
                    }
                }
                // Update the dp table taking the modulo to prevent overflow
                dp[i][j] = (dp[i][j - 1] + count) % MOD;
            }
        }
        // Final answer: sum of all sequences ending at the last digit 'n'
        return dp[n][n];
    }
};

```

Typescript Solution

```typescript
const MOD: number = 1e9 + 7; // Define modulo value for the results to prevent overflow

function numberOfCombinations(num: string): number {
    const n: number = num.length; // Size of the input string
    // 2D array to store the longest common prefix (LCP) lengths between suffixes of 'num'
    let longestCommonPrefix: number[][] = Array.from({length: n + 1}, () => Array(n + 1).fill(0));

    // Preprocess to fill in the longestCommonPrefix array
    for (let i = n - 1; i >= 0; --i) {
        for (let j = n - 1; j >= 0; --j) {
            if (num[i] === num[j]) {
                longestCommonPrefix[i][j] = 1 + longestCommonPrefix[i + 1][j + 1];
            }
        }
    }

    // Comparator function to compare two substrings in 'num' when one is a prefix of the other
    const canPlace: (start: number, prevStart: number, len: number) => boolean = (start, prevStart, len) => {
        const commonPrefixLen: number = longestCommonPrefix[start][prevStart];
        return commonPrefixLen >= len || num[start + commonPrefixLen] >= num[prevStart + commonPrefixLen];
    };

    // 2D array for dynamic programming (dp), where dp[i][j] represents the number of ways
    // to form an increasing sequence up to index i, using a last number of length j
    let dp: number[][] = Array.from({length: n + 1}, () => Array(n + 1).fill(0));
    dp[0][0] = 1; // Base case: 1 way to form a sequence of length 0

    // Populate the dp array
    for (let i = 1; i <= n; ++i) {
        for (let j = 1; j <= i; ++j) {
            let count: number = 0;
            // Check if the current number doesn't start with '0' and if it can be placed in the sequence
            if (num[i - j] !== '0') {
                if (i - j - j >= 0 && canPlace(i - j, i - j - j, j)) {
                    count = dp[i - j][j];
                } else {
                    count = dp[i - j][Math.min(j - 1, i - j)];
                }
            }
            // Update the dp table taking the modulo to prevent overflow
            dp[i][j] = (dp[i][j - 1] + count) % MOD;
        }
    }
    // Final answer: sum of all sequences ending at the last digit 'n'
    return dp[n][n];
}

```

Time and Space Complexity

The time complexity of the provided algorithm is primarily determined by two nested loops that iterate over the length of the input string `num` and the computation of the longest common prefix (LCP) for all substrings. Calculating the LCP takes `O(n^2)` time since it requires two nested loops, each running for `n` iterations, where `n` is the length of the string `num`. Afterward, the main part of the algorithm uses dynamic programming where there are two nested loops that also run for `n` iterations each, thus also taking `O(n^2)` time. In the innermost part of these loops, the `cmp` function is called, which takes constant time due to the precomputed LCP values. Therefore, the dominant part of the time complexity is `O(n^2)`.

The space complexity of the algorithm comes from the storage of the dynamic programming table `dp`, and the longest common prefix table `lcp`, each of size `n x n`. Therefore, the space complexity is `O(n^2)` to store these two tables.