2478. Number of Beautiful Partitions

Hard String Dynamic Programming Leetcode Link

Problem Description

This problem involves devising an algorithm to count how many ways you can divide a string `s` into `k` non-overlapping substrings, where each substring adheres to specific conditions involving their lengths and the digits they start and end with. The string `s` is composed of digits '1' through '9', and the problem specifies additional constraints as follows:

- The string must be partitioned into `k` non-intersecting (non-overlapping) substrings.
- Each resulting substring must be at least `minLength` characters long.
- Each substring must start with a prime digit (2, 3, 5, or 7) and end with a non-prime digit.

The objective is to determine the count of all possible "beautiful partitions" of the string `s`. Due to the potential for a very large number, the result must be returned modulo `10^9 + 7`.

A *substring* should be understood as a sequence of characters that are in consecutive order within the original string `s`.

Intuition

The problem at hand is a dynamic programming challenge. The intuition for such problems generally comes from realizing that the solution for a bigger problem depends on the solution of its sub-problems. Here, we specifically want to find the number of ways to split the string into `k` "beautiful" parts, which will depend on the number of ways to make `j` parts with a subset of the string, where `j < k`.

The solution approach uses two auxiliary two-dimensional arrays (let's call them `f` and `g`) to keep track of the number of ways to partition substrings of `s` into `j` beautiful partitions upto a certain index `i`.

- `f[i][j]` denotes the number of ways we can partition the string upto the `i-th` character resulting in exactly `j` partitions where the `j-th` partition ends at position `i`.
- `g[i][j]` is used to track the running total sum of `f[i][j]` upto that point.

The algorithm incrementally constructs these arrays by iterating over the characters of the string `s` and using the following logic:

1. Initialize both `f[0][0]` and `g[0][0]` to 1, as there is one way to partition an empty string into 0 parts.
2. At each character, `i`, check if a "beautiful" partition could end at this character; that is, the current character must not be a prime, and if we go back `minLength` characters, we should start with a prime (also taking into account edge cases at the ends of the string).
3. If the current character can be the end of a "beautiful" partition, update `f` by using the value from `g` that tracks the number of ways we could have made `j-1` partitions up to this point (since we are potentially adding one more partition here).
4. Update `g` by summing the current values in `g` and `f` for this index, taking care to apply the modulo `10^9 + 7`.

The final answer is the value of `f[n][k]`, as it represents the number of ways to partition the entire string into `k` beautiful partitions.

Solution Approach

The solution implements dynamic programming to count the number of "beautiful" partitions. Let's walk through the critical parts of the implementation:

1. **Initializing Arrays**: Two-dimensional arrays `f` and `g` are created of size `n + 1` by `k + 1`, where `n` is the length of the string `s`. These arrays are initialized to zero but with `f[0][0] = g[0][0] = 1`, because there's exactly one way to partition a string of length 0 into 0 parts.

2. **Dynamic Programming Loop**:

    - Loop through the string `s` while also keeping track of the current position `i` (1-indexed).

    - Check if the current character `c` can be the ending of a "beautiful" partition by ensuring that: a. The substring has at least `minLength` characters, which means `i` should be greater than or equal to `minLength`. b. The character must be non-prime. c. Either we're at the end of the string or the next character is prime.

    - If the substring can end at this character, then for each possible partition count `j` (from 1 to `k`),

        - We calculate `f[i][j]` by looking at `g[i - minLength][j - 1]`, indicating the number of ways to form `j-1` partitions ending before we reach a `minLength` distance from the current index.
3. **Accumulating Results**:

    - The array `g` accumulates the counts of valid partitions. For every index `i` and for every number of partitions `j`, we add the number of ways to extend the existing `j` partitions (`f[i][j]`) to the sum up to the previous character (`g[i - 1][j]`), ensuring that we carry the count forward and also apply the modulo `10^9 + 7` to keep the number within the range.
4. **Returning the Result**:

    - The final result is the value at `f[n][k]`, indicating the number of ways the entire string of length `n` can be partitioned into `k` "beautiful" partitions.

The used data structures here are primarily arrays for dynamic programming. The algorithm itself is a standard dynamic programming pattern, which involves breaking down the problem into smaller subproblems (counting the partitions that can end at each index for a given number of partitions), solving each subproblem (with the loop), and building up to the solution of the overall problem (using the results of subproblems).

Overall, the implementation is efficient as it only requires a single pass through the string and has a time complexity that is linear with respect to the size of the string and the number of partitions, which is O(n * k).

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have the string `s = "23542"` and we want to find out the number of ways to divide this string into `k = 2` non-overlapping substrings with each substring having a minimum length of `minLength = 2`, starting with a prime digit, and ending with a non-prime digit.

Let's walk through the dynamic programming approach step by step.

1. **Initializing Arrays**: We initialize our arrays `f` and `g` with dimensions [6][3] because our string length `n` is 5 (`s.length + 1`) and `k` is 2 (+1 is to include zero partitions). Set all values to 0, except `f[0][0]` and `g[0][0]`, which are set to 1.

```
f = [ [1, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0] ]

g = [ [1, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0] ]
```

2. **Dynamic Programming Loop**:

    - We loop from `i = 1` to `i = 5`, as our string has 5 characters.
    - We check if a "beautiful" substring can end at each `i`:
        - For `i = 1` and `i = 2`, no "beautiful" partition can end as we need at least a substring of length 2.
        - At `i = 3`, we have "235", which ends with a "5" (prime) so it cannot be a "beautiful" partition.
        - At `i = 4`, we have "35" (string[2,3]), which is a valid partition; it starts with "3" (prime) and ends with "5" (non-prime). We set `f[4][1] = g[2][0]`, which is 1.
        - There's no other valid partition at i = 4 since we need to have at least 2 characters in each substring and we are also counting only the partitions which end exactly at the current step.
3. **Accumulating Results**:

    - We update our arrays with the new information:

```
f = [ [1, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 1, 0],
      [0, 0, 0],
      [0, 0, 0] ]

g = [ [1, 0, 0],
      [1, 0, 0],
      [1, 0, 0],
      [1, 1, 0],
      [1, 1, 0],
      [1, 1, 0] ]
```

- For `i = 5`, the substring is "542" which is invalid because it starts with "5" (prime), but "42" (string[3,4]) is validated for the second partition. We set `f[5][2] = g[3][1]`, which is also 1.

4. **Returning the Result**:
    - Finally, we find that `f[5][2]` is 1, which means there is exactly one way to partition the string "23542" into 2 "beautiful" substrings according to the conditions.
    - The result, therefore, is 1 modulo `10^9 + 7`, which is still 1 since it's under the modulo.

In our dynamic arrays:

```
f = [ [1, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 1, 0],
      [0, 0, 0],
      [0, 0, 1] ]
```

`f[n][k]` holds the final answer, which is `f[5][2] = 1`. So there is one beautiful partition of the string "23542" when split into 2 substrings.

Python Solution

```python
class Solution:
    def beautifulPartitions(self, s: str, k: int, minLength: int) -> int:
        # Define primes as a string containing prime digits
        prime_digits = '2357'

        # If the first character of the given string is not a prime digit or
        # the last character is a prime digit, then return 0
        # as it cannot form a beautiful partition
        if s[0] not in prime_digits or s[-1] in prime_digits:
            return 0

        # Define the modulus value for large numbers to prevent overflow
        mod = 10**9 + 7

        # Calculate the length of the string
        n = len(s)

        # Initialize two 2D arrays, f and g, to store intermediate results
        # f[i][j] is the count of beautiful partitions of length exactly i using j partitions
        # g[i][j] includes f[i][j] and also counts partitions ending before i
        f = [[0] * (k + 1) for _ in range(n + 1)]
        g = [[0] * (k + 1) for _ in range(n + 1)]

        # Base case: there's 1 way to divide an empty string using 0 partitions
        f[0][0] = g[0][0] = 1

        # Loop through characters in the string, indexed from 1 for convenience
        for i, char in enumerate(s, 1):
            # Check if the current position can possibly end a partition by checking
            # - it's at least minLength
            # - it's a non-prime digit
            # - next character is prime (or it's the last character)
            if i >= minLength and char not in prime_digits and (i == n or s[i] in prime_digits):
                # If all conditions are satisfied, count this as a potential partition endpoint
                for j in range(1, k + 1):
                    f[i][j] = g[i - minLength][j - 1]

            # Update g[i][j] to include all partitions counted in f[i][j],
            # as well as those that were counted up to position i-1
            for j in range(k + 1):
                g[i][j] = (g[i - 1][j] + f[i][j]) % mod

        # Return the number of beautiful partitions of the whole string using exactly k partitions
        return f[n][k]
```

Java Solution

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

    // Checks if a character is a prime digit
    private boolean isPrimeChar(char c) {
        return c == '2' || c == '3' || c == '5' || c == '7';
    }

    // Returns the number of beautiful partitions of string s
    public int beautifulPartitions(String s, int k, int minLength) {
        int length = s.length();
        // If the first character is not prime or the last character is prime, return 0
        if (!isPrimeChar(s.charAt(0)) || isPrimeChar(s.charAt(length - 1))) {
            return 0;
        }

        // DP table: f[i][j] to store number of ways to partition substring of length i with j beautiful partitions
        int[][] f = new int[length + 1][k + 1];
        // Prefix sum table: g[i][j] to store prefix sums of f up to index i with j beautiful partitions
        int[][] g = new int[length + 1][k + 1];

        // Base case (empty string with 0 partitions)
        f[0][0] = 1;
        g[0][0] = 1;

        // Go through each character in the string
        for (int i = 1; i <= length; ++i) {
            // Check the conditions for forming a beautiful partition
            if (i >= minLength && !isPrimeChar(s.charAt(i - 1)) && (i == length || isPrimeChar(s.charAt(i)))) {
                for (int j = 1; j <= k; ++j) {
                    f[i][j] = g[i - minLength][j - 1]; // Use the value from prefix sum table if the substring is beautiful
                }
            }
            // Update the prefix sum table g with the current values
            for (int j = 0; j <= k; ++j) {
                g[i][j] = (g[i - 1][j] + f[i][j]) % MOD;
            }
        }

        // Return the result from the DP table for full string length with exactly k partitions
        return f[length][k];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    static const int MOD = 1e9 + 7; // Define modulo constant for large numbers

    // Function to count the number of beautiful partitions
    int beautifulPartitions(string s, int partitionCount, int minLength) {
        int strLength = s.size(); // Length of the string

        // Lambda function to check if a character represents a prime digit
        auto isPrime = [](char c) {
            return c == '2' || c == '3' || c == '5' || c == '7';
        };
        // Return 0 if the first character is not prime or the last character is prime
        if (!isPrime(s[0]) || isPrime(s[strLength - 1])) return 0;

        // Dynamic programming tables to keep track of ways to make partitions
        vector<vector<int>> waysToReach(n + 1, vector<int>(partitionCount + 1));
        vector<vector<int>> cumulativeWays(n + 1, vector<int>(partitionCount + 1));

        // Base cases: There is only one way to have no partitions for a string of any length
        waysToReach[0][0] = cumulativeWays[0][0] = 1;

        // Fill the dynamic programming tables
        for (int i = 1; i <= strLength; ++i) {
            if (i >= minLength && !isPrime(s[i - 1]) && (i == strLength || isPrime(s[i]))) {
                for (int j = 1; j <= partitionCount; ++j) {
                    // If a partition ends here, count the ways based on the previous state
                    waysToReach[i][j] = cumulativeWays[i - minLength][j - 1];
                }
            }
            for (int j = 0; j <= partitionCount; ++j) {
                // Update cumulative ways by adding ways up to the current position
                cumulativeWays[i][j] = (cumulativeWays[i - 1][j] + waysToReach[i][j]) % MOD;
            }
        }
        // Return the number of ways to reach the end of the string with exactly k partitions
        return waysToReach[strLength][partitionCount];
    }
};
```

Typescript Solution

```typescript
const MOD = 1e9 + 7; // Define modulo constant for large number computations

// Function to check if a character represents a prime digit
const isPrime = (c: string): boolean => {
    return c === '2' || c === '3' || c === '5' || c === '7';
};

// Function to count the number of beautiful partitions
const beautifulPartitions = (s: string, partitionCount: number, minLength: number): number => {
    const strLength = s.length; // Length of the string

    // Return 0 if the first character is not prime or the last character is prime
    if (!isPrime(s[0]) || isPrime(s[strLength - 1])) return 0;

    // Dynamic programming tables to keep track of ways to make partitions
    const waysToReach: number[][] = Array.from({ length: strLength + 1 }, () => Array(partitionCount + 1).fill(0));
    const cumulativeWays: number[][] = Array.from({ length: strLength + 1 }, () => Array(partitionCount + 1).fill(0));

    // Base case: There is only one way to have no partitions for a string of any length
    waysToReach[0][0] = cumulativeWays[0][0] = 1;

    // Fill the dynamic programming tables
    for (let i = 1; i <= strLength; ++i) {
        if (i >= minLength && !isPrime(s[i - 1]) && (i === strLength || isPrime(s[i]))) {
            for (let j = 1; j <= partitionCount; ++j) {
                // If a partition ends here, count the ways based on the previous state
                waysToReach[i][j] = cumulativeWays[i - minLength][j - 1];
            }
        }
        for (let j = 0; j <= partitionCount; ++j) {
            // Update cumulative ways by adding ways up to the current position
            cumulativeWays[i][j] = (cumulativeWays[i - 1][j] + waysToReach[i][j]) % MOD;
        }
    }
    // Return the number of ways to reach the end of the string with exactly 'partitionCount' partitions
    return waysToReach[strLength][partitionCount];
};
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n * k)`. Here's why:

- The main operation occurs within a double nested loop - the outer loop runs for each character in the input string `s` (with `n` being the length of `s`) and the inner loop runs for `k + 1` times.
- Within the inner loop, each operation is constant time, where we simply check conditions and update the values of `f[i][j]` and `g[i][j]`.
- Since these loops are nested, the total number of operations is the product of the number of iterations of each loop (`n` times for the outer loop, and `k + 1` times for the inner loop), which simplifies to `O(n * k)`.

Space Complexity

The space complexity of the given code is `O(n * k)`. Here's the breakdown:

- Two 2D arrays `f` and `g` are allocated with dimensions `(n + 1) * (k + 1)`. Each array holds `n + 1` rows and `k + 1` columns of integer values, which means they each require `O(n * k)` space.
- Other than the arrays, only a fixed number of integer variables are used, which do not significantly impact the overall space complexity.
- Thus, the dominant factor for space is the size of the 2D arrays, which results in a space complexity of `O(n * k)`.