2266. Count Number of Texts

Medium Hash Table Math String Dynamic Programming Leetcode Link

Problem Description

Alice sends text messages to Bob by pressing keys on a phone keypad where multiple letters are mapped to a single digit. To type a specific letter, Alice must press the corresponding digit key a number of times equal to the position of the letter on that key. However, due to a transmission error, Bob receives a string of digits instead of the text message. The task is to determine the total number of possible original text messages Alice could have sent, given the received string of pressed keys. Note that not all keys are used since the digits '0' and '1' do not map to any letters. The answer could be very large, so it should be returned modulo (10^9 + 7).

For example, the message "bob" would result in Bob receiving "2266622" — '2' pressed three times for 'b', '6' pressed three times for 'o', followed by '2' pressed three times again for the second 'b'.

Intuition

The intuition behind the solution involves recognizing that the task can be tackled as a dynamic programming problem, more specifically, a counting problem. This problem is similar to counting the number of ways to decode a numeric message (like the "Decode Ways" problem on LeetCode), but with the constraint that the sequence of the same digits can represent multiple combinations of letters.

Given the sequence of pressed keys, we can iterate through the sequence and for each contiguous block of identical digits, we calculate the number of ways that block could have been generated by pressing the corresponding letters. The total number of messages is found by multiplying the number of ways for each block, as each block is independent of others.

The sequences of '7' and '9' can have up to four repeated digits because they correspond to four different letters), while the others can have up to three. We can calculate the number of combinations for each case using separate recurrence relations:

- For keys with three letters (digits '2'-'6' and '8'), the current number of combinations (f[n]) is the sum of the last three (f[n-1], f[n-2], f[n-3]).
- For keys with four letters ('7' and '9'), the current number of combinations (g[n]) is the sum of the last four (g[n-1], g[n-2], g[n-3], g[n-4]).

These calculations are done iteratively to fill up the f and g arrays with a precalculated number of combinations for every possible length of digit sequences. Then, in the solution method, we iterate through the `pressedKeys` by grouping the identical adjacent digits together and calculate the answer by multiplying the possible combinations for each group of digits, ensuring to take the modulo at every step to keep the number within bounds.

Solution Approach

The solution implements a dynamic programming approach to solve the problem optimally. The main data structure used is a list or array, which serves as our DP table to store the number of possible combinations for sequences of each acceptable length. The problem also uses the pattern of grouping which is implemented through the `groupby` function from Python's `itertools` module.

Precomputation

In the solution, there are two arrays named `f` and `g` initialized with base cases:

- `f` starts with [1, 1, 2, 4], representing the number of possible texts for keys with three letters.
- `g` starts with [1, 1, 2, 4], representing the number of possible texts for keys '7' and '9' with four letters.

Both arrays are filled up with precomputed values for sequences up to 100,000 characters long. The values are calculated using the recurrent relations mentioned earlier:

- `f[i] = (f[i-1] + f[i-2] + f[i-3]) % mod`
- `g[i] = (g[i-1] + g[i-2] + g[i-3] + g[i-4]) % mod`

The modulo operation is used to ensure our results stay within the bounds of the large prime number (10^9 + 7), as required by the problem statement.

Dynamic Programming Calculation

The core solution is implemented within the `countTexts` method of the `Solution` class:

1. The method initializes `ans` to 1, which will hold the final answer.
2. The input string `pressedKeys` is iterated by grouping the contiguous identical digits together using `groupby`.
3. For each group of identical digits:
   - The length of the group (`m`) is determined by converting the group iterator to a list and taking its length.
   - The character (`ch`) representing the group is checked. If it's '7' or '9', we use array `g` otherwise we use array `f`.
   - We then determine the number of ways this sequence could be generated using `g[m]` or `f[m]`.
   - The total `ans` is updated by multiplying the number of combinations for the current group and taking modulo: `ans = (ans * (g[m] if ch in "79" else f[m])) % mod`.
4. After iterating through all the groups, `ans` is returned providing the total number of possible text messages.

This approach efficiently breaks down the problem by understanding and identifying the independent parts within the input (`pressedKeys`) and leveraging precomputed dynamic programming states to calculate the answer.

Example Walkthrough

Let's say Alice sends a message that results in Bob receiving the following string of pressed keys: "222333". We will use this to illustrate the solution approach.

1. To begin, we parse the input and notice that it consists of groups of the digit '2' followed by the digit '3'.
2. We then initialize our dynamic programming arrays `f` and `g` with precomputed values.
   - For simplicity, let's assume we have only computed these arrays for lengths up to 5: `f = [1, 1, 2, 4, 7]` and `g = [1, 1, 2, 4, 8]`. These arrays represent the number of ways we can generate a string with three characters (for `f`) and four characters (for `g`) assigned a single key, for strings of different lengths.
3. We start with an answer (`ans`) of 1.
4. We group the pressed keys into contiguous identical digits using `groupby`. In our case, we have two groups: '222' and '33'.
5. We iterate through each group and determine the number of combinations for each:
   - For the first group '222' (which has three identical digits):
     - The length of the group (`m`) is 3.
     - Since the digit `2` corresponds to keys with three letters, we use array `f`.
     - Looking up `f[3]`, we get 2, meaning there are 2 ways to generate the sequence '222'.
     - So we update our answer: `ans = ans * f[3] % mod = 1 * 2 % mod = 2`.
   - For the second group '33' (which has two identical digits):
     - The length of the group (`m`) is 2.
     - Since the digit `3` corresponds to keys with three letters, we use array `f`.
     - Looking up `f[2]`, we get 1, meaning there is 1 way to generate the sequence '33'.
     - We update our answer again: `ans = ans * f[2] % mod = 2 * 1 % mod = 2`.
6. After accounting for all groups, our final answer (`ans`) is 2, meaning there are 2 possible text messages Alice could have sent to result in the received string "22233".

In this example, the possible original messages could be "abdf" (where '2' is pressed once for each letter) followed by "dff" (where '3' is pressed once for each letter), or "abc" followed by "ee" (where '3' is pressed twice for 'e'). Therefore, the total number of possible original text messages is indeed 2.

Python Solution

```python
from itertools import groupby

MOD = 10**9 + 7  # Define the modulo for large numbers to ensure the result fits in specified range.

# These lists will hold the number of ways to press keys
# f for keys that have 3 letters (i.e., 2, 3, 4, 5, 6, 8)
# g for keys that have 4 letters (i.e., 7, 9)
f = [1, 1, 2, 4]  # Base cases for f sequences
g = [1, 1, 2, 4]  # Base cases for g sequences

# Precompute the answers for up to 100000 sequences (which is more than enough for our purpose)
for i in range(4, 100000):
    f.append((f[-1] + f[-2] + f[-3]) % MOD)  # Use modulo operation to keep the number within limits
    g.append((g[-1] + g[-2] + g[-3] + g[-4]) % MOD)  # Include an additional term for 4-letter keys

class Solution:
    def countTexts(self, pressedKeys: str) -> int:
        # Calculating the number of different text messages that can be created based on a string of pressed keys.
        ans = 1  # Initialize answer to 1 to start multiplying the individual counts.

        # Group by each unique sequence of key presses
        for key_char, group in groupby(pressedKeys):
            press_count = len(list(group))  # Count the number of times the key is pressed consecutively.

            # Calculate different ways to press the sequence of keys and update the answer
            if key_char in "79":
                ans = ans * g[press_count] % MOD  # Use g for keys with 4 letters
            else:
                ans = ans * f[press_count] % MOD  # Use f for keys with 3 letters

        # Return the total number of different texts that could be typed
        return ans
```

Java Solution

```java
class Solution {

    // Constants
    private static final int MAX_KEY_PRESS = 100010; // Maximum length of the dynamic programming arrays
    private static final long MOD = (long)1e9 + 7; // Modulus value for avoiding integer overflow

    // Dynamic programming arrays to store intermediate results
    private static long[] pressSequence3 = new long[MAX_KEY_PRESS]; // Stores results for keys with 3 letters
    private static long[] pressSequence4 = new long[MAX_KEY_PRESS]; // Stores results for keys with 4 letters

    // Static block for pre-computing results used in the countTexts function
    static {
        pressSequence3[0] = 1;
        pressSequence3[1] = 1;
        pressSequence3[2] = 2;
        pressSequence3[3] = 4;

        pressSequence4[0] = 1;
        pressSequence4[1] = 1;
        pressSequence4[2] = 2;
        pressSequence4[3] = 4;

        // Filling dynamic programming arrays with the possible combinations
        for (int i = 4; i < MAX_KEY_PRESS; ++i) {
            // For keys '2', '3', '4', '5', '6', '8'
            pressSequence3[i] = (pressSequence3[i - 1] + pressSequence3[i - 2] + pressSequence3[i - 3]) % MOD;
            // For keys '7', '9'
            pressSequence4[i] = (pressSequence4[i - 1] + pressSequence4[i - 2] + pressSequence4[i - 3] + pressSequence4[i - 4]) % MOD;
        }
    }

    public int countTexts(String pressedKeys) {
        long totalCombinations = 1; // Initialize the total combination count
        for (int i = 0, n = pressedKeys.length(); i < n; ++i) {
            int j = i + 1;
            // Count the number of times the current key is consecutively pressed
            while (j < n && pressedKeys.charAt(j) == pressedKeys.charAt(i)) {
                ++j;
            }
            int pressCount = j - i + 1; // Total presses for the current key

            // Calculate combinations based on the current key being either a 3-letter or 4-letter key
            if (pressedKeys.charAt(i) == '7' || pressedKeys.charAt(i) == '9') {
                // For keys '7' and '9', use pressSequence4 array
                totalCombinations = totalCombinations * pressSequence4[pressCount] % MOD;
            } else {
                // For other keys, use pressSequence3 array
                totalCombinations = totalCombinations * pressSequence3[pressCount] % MOD;
            }
            i = j; // Move index to the last occurrence of the currently pressed key
        }
        return (int) totalCombinations; // Return the total count as an integer
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
    // Constants
    static constexpr int MAX_KEY_PRESS = 100010; // Maximum length of the dynamic programming arrays
    static constexpr int MOD = 1000000007; // Modulus value for avoiding integer overflow

    // Dynamic programming arrays to store intermediate results
    static std::vector<long long> pressSequence3; // Stores results for keys with 3 letters
    static std::vector<long long> pressSequence4; // Stores results for keys with 4 letters

    // Static member initialization using lambda
    static std::vector<long long> initVector(int size) {
        std::vector<long long> vec(size, 0);
        vec[0] = 1;
        vec[1] = 1;
        vec[2] = 2;
        vec[3] = 4;
        for (int i = 4; i < size; ++i) {
            vec[i] = (vec[i - 1] + vec[i - 2] + vec[i - 3]) % MOD;
            if (i >= 4) // For the 4-key sequence
                vec[i] = (vec[i] + vec[i - 4]) % MOD;
        }
        return vec;
    }

public:
    // Constructor
    Solution() {
        pressSequence3 = std::vector<long long>(MAX_KEY_PRESS); // Only the first 3 terms differ between pre
        pressSequence4 = std::vector<long long>(MAX_KEY_PRESS); // Hence initializing once using a function
        pressSequence3 = initVector(MAX_KEY_PRESS);
        pressSequence4 = pressSequence3;
    }

    // Function to count the number of distinct texts that can be generated from the input string of pressed keys
    int countTexts(std::string pressedKeys) {
        long long totalCombinations = 1; // Initialize the total combination count
        for (int i = 0, n = pressedKeys.length(); i < n; ++i) {
            int j = i + 1;
            char currentKey = pressedKeys[i];
            // Count the number of times the current key is consecutively pressed
            while (j < n && pressedKeys[j] == currentKey) {
                ++j;
            }
            int pressCount = j - i + 1; // Total presses for the current key

            // Calculate combinations based on the current key being either a 3-letter or 4-letter key
            if (currentKey == '7' || currentKey == '9') {
                // For keys '7' and '9', use pressSequence4 array
                totalCombinations = totalCombinations * pressSequence4[pressCount] % MOD;
            } else {
                // For other keys, use pressSequence3 array
                totalCombinations = totalCombinations * pressSequence3[pressCount] % MOD;
            }
            i = j; // Move index to the last occurrence of the currently pressed key
        }
        return static_cast<int>(totalCombinations); // Return the final count as an integer
    }
};

// Static member definitions
std::vector<long long> Solution::pressSequence3 = {};
std::vector<long long> Solution::pressSequence4 = {};
```

Typescript Solution

```typescript
// Constants
const MAX_KEY_PRESS = 100010; // Maximum length of the dynamic programming arrays
const MOD = 1e9 + 7; // Modulus value for avoiding integer overflow

// Dynamic programming arrays to store intermediate results
let pressSequence3: number[] = new Array(MAX_KEY_PRESS).fill(0); // Stores results for keys with 3 letters
let pressSequence4: number[] = new Array(MAX_KEY_PRESS).fill(0); // Stores results for keys with 4 letters

// Pre-computing results used in the countTexts function
pressSequence3[0] = 1;
pressSequence3[1] = 1;
pressSequence3[2] = 2;
pressSequence3[3] = 4;

pressSequence4[0] = 1;
pressSequence4[1] = 1;
pressSequence4[2] = 2;
pressSequence4[3] = 4;

// Filling dynamic programming arrays with the possible combinations
for (let i = 4; i < MAX_KEY_PRESS; ++i) {
    // For keys '2', '3', '4', '5', '6', '8'
    pressSequence3[i] = (pressSequence3[i - 1] + pressSequence3[i - 2] + pressSequence3[i - 3]) % MOD;
    // For keys '7', '9'
    pressSequence4[i] = (pressSequence4[i - 1] + pressSequence4[i - 2] + pressSequence4[i - 3] + pressSequence4[i - 4]) % MOD;
}

function countTexts(pressedKeys: string): number {
    let totalCombinations = 1; // Initialize the total combination count
    for (let i = 0, n = pressedKeys.length; i < n; ++i) {
        let j = i + 1;
        const currentKey = pressedKeys.charAt(i);
        // Count the number of times the current key is consecutively pressed
        while (j < n && pressedKeys.charAt(j) == currentKey) {
            ++j;
        }
        const pressCount = j - i + 1; // Total presses for the current key

        // Calculate combinations based on the current key being either a 3-letter or 4-letter key
        if (currentKey == '7' || currentKey == '9') {
            // For keys '7' and '9', use pressSequence4 array
            totalCombinations = totalCombinations * pressSequence4[pressCount] % MOD;
        } else {
            // For other keys, use pressSequence3 array
            totalCombinations = totalCombinations * pressSequence3[pressCount] % MOD;
        }
        i = j; // Move index to the last occurrence of the currently pressed key
    }
    return totalCombinations; // Return the final count as an integer
}

// You can now call countTexts with a string of pressed keys.
```

Time and Space Complexity

The given Python code implements a dynamic programming solution to count the possible ways to interpret a string of pressed keys. It initializes precomputed lists `f` and `g` for memoization and then implements the function `countTexts` to compute the result, using these lists.

Time Complexity:

- The pre-computation of `f` and `g` lists is done in a loop that runs for a fixed number of 100,000 iterations. Within each iteration, the code performs a constant number of mathematical operations and list accesses, resulting in a time complexity of $O(1)$ for each iteration. Altogether, the pre-computation of `f` and `g` has a time complexity of $O(100000)$ which is considered a constant and hence can be simplified to $O(1)$.
- The `countTexts` function iterates over `pressedKeys` once, and for each unique character, it performs a multiplication and a modulo operation. The complexity of these operations is $O(1)$. If n is the length of the `pressedKeys` string, the time complexity of this part is $O(n)$.
- The `groupby` function has a time complexity of $O(n)$ as well, since it has to iterate through the entire `pressedKeys` string once.

Combining the pre-computation and the `countTexts` method, the overall time complexity of the entire code is $O(n)$.

Space Complexity:

- The `f` and `g` lists are of fixed size `100,004` each, and they don't scale with the input, so their space requirement is $O(1)$ in the context of this problem.
- The `countTexts` function uses additional space for the output of `groupby` and the temporary list created for each group `x`. In the worst-case scenario (e.g., all characters are the same), the space complexity could be $O(n)$. However, since only one group is stored in memory at a time, this additional space is also considered $O(1)$.
- Thus, the additional space complexity due to the `countTexts` function is $O(n)$.

Final Space Complexity is therefore $O(n)$ when considering the storage required for input processing in the `countTexts` function.