2266. Count Number of Texts

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

Hash Table

Medium

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 that could result in 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

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.

"Decode Ways" problem on LeetCode), but with the constraint that the sequence of the same digits can represent multiple

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]).

The sequences of '7' and '9' can have up to four repeated digits (because they correspond to four different letters), while the others

• 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 sequence. 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
- Solution Approach The solution implements a dynamic programming approach to solve the problem optimally. The main data structure used is a list or

an array, which serves as our DP table to store the number of 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.

recurrent relations mentioned earlier:

Precomputation

every step to keep the number within bounds.

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

• f[i] = (f[i-1] + f[i-2] + f[i-3]) % mod

1. The method initializes ans to 1, which will hold the final answer.

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

Dynamic Programming During Execution

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

2. The input string pressedKeys is iterated by grouping the contiguous identical digits together using groupby.

• 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.

∘ 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.

3. For each group of identical digits:

illustrate the solution approach.

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.

• We then determine the number of ways this sequence could be generated using g[m] or f[m].

Example Walkthrough

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.

5. We iterate through each group and determine the number of combinations for each:

Since the digit 2 corresponds to keys with three letters, we use array f.

• So we update our answer: ans = ans * f[3] % mod = 1 * 2 % mod = 2.

Looking up f[3], we get 2, meaning there are 2 ways to generate the sequence '222'.

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

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

private static final int MAX_KEY_PRESS = 100010; // Maximum length of the dynamic programming arrays

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

pressSequence3[i] = (pressSequence3[i - 1] + pressSequence3[i - 2] + pressSequence3[i - 3]) % MOD;

pressSequence4[i] = (pressSequence4[i - 1] + pressSequence4[i - 2] + pressSequence4[i - 3] + pressSequence4[i - 4]) % M

private static final int MOD = (int) 1e9 + 7; // Modulus value for avoiding integer overflow

answer = 1 # Initialize answer to 1, as we'll multiply the individual counts.

Return the total number of different texts that could be typed

// Static block for pre-computing results used in the countTexts function

// Filling dynamic programming arrays with the possible combinations

long totalCombinations = 1; // Initialize the total combination count

For the first group '222' (which has three identical digits):

For the second group '33' (which has two identical digits):

■ The length of the group (m) is 3.

The length of the group (m) is 2.

These lists will hold the number of ways to press keys

def countTexts(self, pressedKeys: str) -> int:

Group by each unique sequence of key presses

// Dynamic programming arrays to store intermediate results

1, 2, 4, 8, 15]. These arrays represent the number of ways we can generate a string with three characters (for f) and four characters (for g) mapped to 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'.

∘ For simplicity, let's assume we have only computed these arrays for lengths up to 5: f = [1, 1, 2, 4, 7, 13] and g = [1,

Let's say Alice sends a message that results in Bob receiving the following string of pressed keys: "22233". We will use this to

Since the digit 3 corresponds to keys with three letters, we use array f.

original text messages is indeed 2.

1 from itertools import groupby

Python Solution

12 for _ in range(100000):

16 class Solution:

13

14

15

17

18

19

20

21

30

31

32

33

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

result in the received string "22233".

 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

- In this example, the possible original messages could be "abc" (where '2' is pressed once for each letter) followed by "df" (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
- 6 # f for keys that have 3 letters (i.e., 2, 3, 4, 5, 6, 8) 7 # g for keys that have 4 letters (i.e., 7 and 9) 8 f = [1, 1, 2, 4] # Base cases for f sequence 9 g = [1, 1, 2, 4] # Base cases for g sequence 10 11 # Precompute the answers for up to 100000 presses (which is more than enough for our purpose)

22 for key_char, group in groupby(pressedKeys): 23 press_count = len(list(group)) # Count the number of times the key is pressed consecutively 24 # Calculate different ways to press the sequence of keys and update the answer 25 26 if key_char in "79": 27 answer = (answer * g[press_count]) % MOD # Use g for keys with 4 letters else: 28 29 answer = (answer * f[press_count]) % MOD # Use f for keys with 3 letters

"""Calculates the number of different text messages that can be created based on a string of pressed keys."""

```
1 class Solution {
3
      // Constants
```

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;

int j = i;

// For keys '7', '9'

public int countTexts(String pressedKeys) {

for (int i = 4; i < MAX_KEY_PRESS; ++i) {</pre>

// For keys '2', '3', '4', '5', '6', '8'

for (int i = 0, n = pressedKeys.length(); i < n; ++i) {</pre>

for (size_t i = 0, n = pressedKeys.length(); i < n; ++i) {</pre>

if (currentKey == '7' || currentKey == '9') {

while $(j + 1 < n \&\& pressedKeys[j + 1] == currentKey) {$

// For keys '7' and '9', use pressSequence4 array

// For other keys, use pressSequence3 array

2 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

// Pre-computing results used in the countTexts function

// Count the number of times the current key is consecutively pressed

// Calculate combinations based on the current key being either a 3-letter or 4-letter key

totalCombinations = (totalCombinations * pressSequence4[pressCount]) % MOD;

totalCombinations = (totalCombinations * pressSequence3[pressCount]) % MOD;

i = j; // Move index to the last occurrence of the currently pressed key

6 let pressSequence3: number[] = new Array(MAX_KEY_PRESS); // Stores results for keys with 3 letters

let pressSequence4: number[] = new Array(MAX_KEY_PRESS); // Stores results for keys with 4 letters

return static_cast<int>(totalCombinations); // Return the final count as an integer

int pressCount = j - i + 1; // Total presses for the current key

char currentKey = pressedKeys[i];

64 std::vector<long long> Solution::pressSequence3 = {};

65 std::vector<long long> Solution::pressSequence4 = {};

size_t j = i;

++j;

} else {

63 // Static member definitions

Typescript Solution

10 pressSequence3[0] = 1;

11 pressSequence3[1] = 1;

Java Solution

return answer

```
36
                 char currentKey = pressedKeys.charAt(i);
 37
                 // Count the number of times the current key is consecutively pressed
                 while (j + 1 < n && pressedKeys.charAt(j + 1) == currentKey) {</pre>
 38
 39
                     ++j;
 40
 41
                 int pressCount = j - i + 1; // Total presses for the current key
 42
 43
                 // Calculate combinations based on the current key being either a 3-letter or 4-letter key
                 if (currentKey == '7' || currentKey == '9') {
 44
                     // For keys '7' and '9', use pressSequence4 array
 45
 46
                     totalCombinations = totalCombinations * pressSequence4[pressCount];
 47
                 } else {
 48
                     // For other keys, use pressSequence3 array
 49
                     totalCombinations = totalCombinations * pressSequence3[pressCount];
 50
                 totalCombinations %= MOD; // Modulo to prevent overflow
 51
 52
                 i = j; // Move index to the last occurrence of the currently pressed key
 53
 54
             return (int) totalCombinations; // Return the final count as an integer
 55
 56 }
 57
C++ Solution
   #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
  8
  9
         // Dynamic programming arrays to store intermediate results
         static std::vector<long long> pressSequence3; // Stores results for keys with 3 letters
 10
         static std::vector<long long> pressSequence4; // Stores results for keys with 4 letters
 11
 12
 13
         // Static member initialization
 14
         static std::vector<long long> initVector(int size) {
             std::vector<long long> v(size, 0);
 15
 16
             v[0] = 1;
 17
             v[1] = 1;
 18
             v[2] = 2;
 19
             v[3] = 4;
 20
             for (int i = 4; i < size; ++i) {
                 v[i] = (v[i-1] + v[i-2] + v[i-3]) % MOD;
 21
 22
                 if (i >= 4) { // For the 4-key sequence
 23
                     v[i] = (v[i] + v[i - 4]) % MOD;
 24
 25
 26
             return v;
 27
 28 public:
 29
        // Constructor
 30
         Solution() {
             static std::vector<long long> init_pressSequence3 = initVector(MAX_KEY_PRESS); // Only the first 3 terms differ between pre
 31
 32
             pressSequence3 = init_pressSequence3;
             static std::vector<long long> init_pressSequence4 = initVector(MAX_KEY_PRESS); // Hence initializing once using a function
 33
 34
             pressSequence4 = init_pressSequence4;
 35
 36
         // Function to count the number of distinct texts that can be generated from the input string of pressed keys
 37
 38
         int countTexts(std::string pressedKeys) {
 39
             long long totalCombinations = 1; // Initialize the total combination count
```

12 pressSequence3[2] = 2; 13 pressSequence3[3] = 4; 14

1 // Constants

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

62

66

8

61 };

```
15 pressSequence4[0] = 1;
 16 pressSequence4[1] = 1;
 17 pressSequence4[2] = 2;
 18 pressSequence4[3] = 4;
 19
 20 // Filling dynamic programming arrays with the possible combinations
 21 for (let i = 4; i < MAX_KEY_PRESS; ++i) {
        // For keys '2', '3', '4', '5', '6', '8'
 22
         pressSequence3[i] = (pressSequence3[i - 1] + pressSequence3[i - 2] + pressSequence3[i - 3]) % MOD;
 23
        // For keys '7', '9'
 24
 25
         pressSequence4[i] = (pressSequence4[i - 1] + pressSequence4[i - 2] + pressSequence4[i - 3] + pressSequence4[i - 4]) % MOD;
 26 }
 27
    // Function to count the number of distinct text messages that can be created
     function countTexts(pressedKeys: string): number {
         let totalCombinations: number = 1; // Initialize the total combination count
 30
 31
         for (let i = 0, n = pressedKeys.length; i < n; ++i) {</pre>
 32
 33
             let j = i;
             const currentKey = pressedKeys.charAt(i);
 34
 35
             // Count the number of times the current key is consecutively pressed
             while (j + 1 < n && pressedKeys.charAt(j + 1) === currentKey) {</pre>
 36
 37
                 ++j;
 38
 39
             const pressCount = j - i + 1; // Total presses for the current key
 40
             // Calculate combinations based on the current key being either a 3-letter or 4-letter key
 41
 42
             if (currentKey === '7' || currentKey === '9') {
 43
                 // For keys '7' and '9', use pressSequence4 array
                 totalCombinations *= pressSequence4[pressCount];
 44
 45
             } else {
                 // For other keys, use pressSequence3 array
 46
 47
                 totalCombinations *= pressSequence3[pressCount];
 48
 49
             totalCombinations %= MOD; // Apply modulo to prevent overflow
             i = j; // Move index to the last occurrence of the currently pressed key
 50
 51
 52
 53
         return totalCombinations; // Return the final count
 54
 55
    // You can now call countTexts with a string of pressed keys.
 57
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 0(1) for each iteration. Altogether, the pre-computation of f and g has a time complexity of 0(100000) which is considered a constant

and hence can be simplified to 0(1).

Space Complexity:

• 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 0(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).

• 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 0(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 s. 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

 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.

stored in memory at a time, this does not add up across different groups.