<https://leetcode.com/problems/number-of-substrings-with-only-1s/description/>

Given a binary string s, return the number of substrings with all characters 1's. Since the answer may be too large, return it modulo 10^9 + 7.

**Example 1:**

**Input:** s = "0110111"

**Output:** 9

**Explanation:** There are 9 substring in total with only 1's characters."1" -> 5 times."11" -> 3 times."111" -> 1 time.

**Example 2:**

**Input:** s = "101"

**Output:** 2

**Explanation:** Substring "1" is shown 2 times in s.

**Example 3:**

**Input:** s = "111111"

**Output:** 21

**Explanation:** Each substring contains only 1's characters.

**Constraints:**

1 <= s.length <= 10^5

s[i] is either '0' or '1'.

**Attempt 1: 2024-01-05**

**Solution 1: Math (10 min)**

class Solution {

    public int numSub(String s) {

        int MOD = (int)(1e9 + 7);

        int total\_count = 0;

        int curr\_section\_one\_count = 0;

        for(int i = 0; i < s.length(); i++) {

            if(s.charAt(i) == '1') {

                curr\_section\_one\_count++;

                total\_count = (total\_count + curr\_section\_one\_count) % MOD;

            } else {

                curr\_section\_one\_count = 0;

            }

        }

        return total\_count;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/number-of-substrings-with-only-1s/solutions/731580/java-c-python-count/>

**Explanation**

count the current number of consecutive "1".

For each new element, there will be more count substring, with all characters 1's.

public int numSub(String s) {

int res = 0, count = 0, mod = (int)1e9 + 7;

for (int i = 0; i < s.length(); ++i) {

count = s.charAt(i) == '1' ? count + 1 : 0;

res = (res + count) % mod;

}

return res;

}

**Refer to**

<https://leetcode.com/problems/number-of-substrings-with-only-1s/solutions/731558/python-easy-readable-code/>

The example would be self explanatoy:

"0110111" can be evaluated to -->

0120123

Now if we sum up all these digits:

1+2+1+2+3 = 9 is the result!

class Solution(object):

def numSub(self, s):

res, currsum = 0,0

for digit in s:

if digit == '0':

currsum = 0

else:

currsum += 1

res+=currsum

return res % (10\*\*9+7)

**Refer to**

<https://algo.monster/liteproblems/1513>

**Problem Description**

The problem requires us to find and count the number of substrings within a given binary string s that consist entirely of the character '1'. For example, in the string "0110111", there are several such substrings: "1", "11", "111", "11", "1", and each of them should be counted. The challenge here is to do this efficiently and to deal with potentially very large numbers, the counts need to be returned modulo 10^9 + 7, which is a large prime number commonly used to prevent integer overflow issues in competitive programming.

**Intuition**

The intuition behind the solution is based on the observation that for a sequence of '1's of length n, there are exactly n \* (n + 1) / 2 substrings that can be formed, all of which consist solely of '1's. **This is because each additional '1' potentially adds as many new substrings as its position in the sequence (first '1' adds 1 substring, second '1' adds 2 substrings, and so on). So, we are essentially counting the lengths of continuous sequences of '1's and calculating the number of substrings for each sequence.**

The solution involves iterating through the string and using a counter variable cnt to keep track of the length of a consecutive sequence of '1's. When we encounter a '1', we increment cnt. If we encounter a '0', we reset cnt to zero, because it breaks the sequence of '1's. After each step, regardless of whether we're continuing a sequence or ending one, we add the current value of cnt to ans, which accumulates the total number of substrings of '1's we've seen so far.

Finally, since the answer could be a very large number, we return the total count modulo 10^9 + 7 as instructed by the problem description.

**Solution Approach**

The implementation of the solution is relatively straightforward with no need for complex data structures; the only variable that holds state during iteration is cnt, which counts the length of consecutive '1's. With every step forward in the string, the algorithm performs the following actions:

Check the current character (c) of the string.

If c is '1', increment the cnt variable: this increases the length of the current sequence of '1's by one.

If c is not '1' (in this case, it could only be '0' since it's a binary string), reset cnt to zero. This indicates the end of the current sequence of '1's and resets the count for the next sequence.

After updating cnt, add its current value to ans. This step accumulates the total number of valid substrings formed by the sequences of '1's encountered so far.

This process works because each time a '1' is encountered and cnt is incremented, it effectively counts all the new substrings that could end with this '1'. If you have a sequence of length n, then adding one more '1' to the sequence adds n + 1 substrings: all the previous substrings plus a new substring that includes the entire sequence.

The use of the modulo operation % (10\*\*9 + 7) ensures that we handle the possibility of very large outputs, preventing integer overflow and keeping the result within the limits of the problem's constraints.

No additional data structures are required, and the pattern used in this algorithm is commonly known as a single pass or linear scan, as it requires only one iteration through the input string to compute the result.

**Solution Implementation**

class Solution {

// This method counts the number of substrings that contains only the character '1'

public int numSub(String s) {

final int MODULO = (int) 1e9 + 7; // Define a constant for modulo operation

int countSubstrings = 0; // Store the count of valid substrings

int consecutiveOnes = 0; // Counter for consecutive '1's

// Iterate through the string character by character

for (int i = 0; i < s.length(); ++i) {

// If the current character is '1', increment the consecutive ones counter

// Otherwise, reset it to zero since we're only interested in consecutive '1's

consecutiveOnes = s.charAt(i) == '1' ? consecutiveOnes + 1 : 0;

// Add the current count of consecutive '1's to countSubstrings

// This works because for each new '1' character, it forms a new substring

// ending at this character that has not been counted yet

countSubstrings = (countSubstrings + consecutiveOnes) % MODULO;

}

// Return the total number of substrings found, modulo the defined constant

return countSubstrings;

}

}

**Time and Space Complexity**

The code provided counts the number of substrings of contiguous '1's in a given string s. The time complexity and space complexity of the algorithm are analyzed below.

**Time Complexity**

The algorithm goes through each character of the input string s exactly once. During this process, it performs constant time operations such as checking if a character is '1' or '0', incrementing the cnt variable, and adding the value of cnt to ans. Because these operations do not depend on the size of the string and are repeated for each character, the time complexity is directly proportional to the length of the string. Therefore, the time complexity is O(n), where n is the length of the input string s.

**Space Complexity**

The space used by the algorithm includes a fixed number of integer variables ans and cnt, which do not scale with the size of the input string. As a result, the algorithm uses a constant amount of additional space, resulting in a space complexity of O(1), indicating that it is independent of the input size.

**Refer to**

[L1759.Count Number of Homogenous Substrings](note://WEB97d08a11b7c9da580d01c75e25918b5a)

[L2062.Count Vowel Substrings of a String (Ref.L2461)](note://WEBffe1facb0ba61c1f4e6681e23ad781cb)