1987. Number of Unique Good Subsequences

String) **Dynamic Programming** Hard

Problem Description

leading zeros. Our task is to find how many such unique good subsequences exist in the given binary string. We need to calculate this count modulo 10⁹ + 7 due to the potential for very large numbers.

The problem requires us to count the unique good subsequences in a given binary string binary. A subsequence is a series of

characters that can be derived from the original string by removing zero or more characters without changing the order of the

remaining characters. A good subsequence is one that does not have leading zeros, except for the single "0". For example, in

"001", the good subsequences are "0", "0", and "1", but subsequences like "00", "01", and "001" are not good because they have

To solidify the understanding, let's consider an example: If binary = "101", the unique good subsequences are "", "1", "0", "10", "11", "101". Note that "" (the empty string) is not considered a good subsequence since it's not non-empty, so the answer would

be 5 unique good subsequences.

ntuition

To solve this problem, we leverage dynamic programming to keep track of the count of unique good subsequences ending with

Here's the intuition for the solution:

• If we encounter a '0' in the binary string, we can form new unique subsequences by appending '0' to all the unique good subsequences ending with '1'. However, because '0' cannot be at the leading position, we cannot start new subsequences with it.

• If we encounter a '1', we can form new unique subsequences by appending it to all the unique good subsequences ending in '0' and '1', and we

can also start a new subsequence with '1'.

unique good subsequences.

'0' and '1', separately.

- Special care must be taken to include the single "0" subsequence if '0' is present in the string, which is handled by the ans = 1 line if a '0' is found. The variable f holds the number of unique good subsequences ending with '1', and g with '0'. When iterating through the string,
- if we find '0', we update g. If it's '1', we update f. The variable ans is set to '1' when we find a '0' to account for the single "0" subsequence. Finally, we sum up the two variables (and add 'ans' if '0' was encountered) to get the total count and take the
- modulo $10^9 + 7$. By keeping track of these two counts, we can update how many subsequences we have as we iterate through the binary string.

Solution Approach

represent the count of unique good subsequences ending in '1' and '0', respectively.

• Finally, we return ans % mod, ensuring that the result adheres to the modulo constraint.

The implementation involves iterating through each character in the binary string and updating two variables, f and g, which

The <u>dynamic programming</u> approach ensures that we do not count duplicates, and by taking the sum of f and g, we cover all

• Initialize f and g to zero. f is used to keep track of subsequences ending with '1', and g for those ending with '0'. • The ans variable is initialized to zero, which will later be used to ensure that we include the single "0" subsequence if necessary.

• We also define mod = 10***9 + 7 to handle the modulo operation for a large number of good subsequences. • We loop through each character c in the binary string:

subsequence.

o If c is "0", we update g to be g + f. This counts any new subsequences formed by appending '0' to subsequences that were previously ending with '1'. We cannot start a new subsequence with '0' to prevent leading zeros. We also set ans to 1 to account for the "0"

∘ If c is "1", we update f to be f + g + 1. Here, f + g accounts for the new subsequences ending with '1' formed by appending '1' to all

The algorithm uses dynamic programming efficiently, as it effectively keeps tally of the subsequences without generating them,

These update rules succinctly represent the logic needed to arrive at the solution of counting the number of unique good

previous subsequences ending with '0' and '1'. The + 1 handles the case where '1' itself can start a new good subsequence. • After the loop, ans is updated to be the sum of g, f, and the existing ans, which accounts for the "0" when it is present.

Here is a step-by-step guide through the implementation:

Here is the mathematical representation of the update steps, enclosed in backticks for proper markdown display:

preventing duplicate counts, and cleverly uses modular arithmetic to manage the potentially large numbers.

- subsequences in the binary string.
- Let's walk through a small example to illustrate the solution approach with the binary string binary = "00110". 1. Initialize the variables:
- f = 0 // Number of subsequences ending with '1'.

2. Start with the first character, '0':

Results: f = 0, g = 0, ans = 1.

• Results: f = 0, g = 0, ans = 1.

• g = 0 // Number of subsequences ending with '0'.

• ans = 0 // To account for the single "0" subsequence.

But we set ans to 1 to account for the "0" subsequence.

Again, it's '0', so we follow the same step, g remains as it is.

Since it's '0', we cannot start a new subsequence, so g remains 0.

• When c is "0": g = (g + f) % mod

• Finally: ans = (ans + f + g) % mod

• When c is "1": $f = (f + g + 1) \% \mod$

mod = 10**9 + 7 // For modulo operation.

Example Walkthrough

3. Move to the second character, '0':

• The character '1' allows new subsequences by appending '1' to all good subsequences ending with '0' and '1', plus a new subsequence just '1'.

We find there are 5 unique good subsequences in the binary string "00110". These are "0", "1", "10", "11", and "110". Thus, the

4. Continue with the third character, '1':

5. Next character, '1':

• Update $f: f = (f + g + 1) \% \mod$. • Results: f = 1, g = 0, ans = 1.

• Update g: g = (g + f) % mod = (0 + 2) % mod = 2.

• ans = (ans + f + g) % mod = (1 + 2 + 2) % mod = 5.

answer agrees with our algorithm's final computation.

 Now, f will include the previous f, plus g, plus a new subsequence '1': • Update $f: f = (1 + 0 + 1) \% \mod = 2$.

• The value of ans doesn't change because we only include the single "0" once.

6. Last character, '0': • This '0' is appended to good subsequences ending with '1', which are counted in f.

And since it's '0', ans stays as 1.

• Results: f = 2, g = 2, ans = 1.

Solution Implementation

count ones = 0

count zeros = 0

mod = 10**9 + 7

else:

if has zero:

has_zero = True

return total_unique_subseq

has_zero = False

• Results: f = 2, g = 0, ans = 1.

- Now we add up f, g, and ans for the total count of unique good subsequences:
- **Python** class Solution:

'count ones' stores the count of subsequences ending with '1'.

'has zero' tracks if a subsequence with '0' has been observed.

count_zeros = (count_zeros + count_ones) % mod

count_ones = (count_ones + count_zeros + 1) % mod

total_unique_subseq = (total_unique_subseq + 1) % mod

public int numberOfUniqueGoodSubsequences(String binary) {

// Loop through each character in the binary string

function numberOfUniqueGoodSubsequences(binary: string): number {

// Iterate over each character in the binary string.

uniqueGoodSubsequencesCount = 1;

def numberOfUniqueGoodSubsequences(self, binary: str) -> int:

Modular constant to avoid overflow on large numbers.

Iterate through the given binary string character by character.

count_zeros = (count_zeros + count_ones) % mod

count ones = (count ones + count zeros + 1) % mod

total_unique_subseq = (total_unique_subseq + 1) % mod

Aggregate total unique subsequences from those ending in '1' and '0'.

We've encountered a zero, so we note that.

total_unique_subseq = (count_ones + count_zeros) % mod

Update count zeros with count ones since every subsequence

Update count ones with count zeros since every subsequence

If we've encountered a zero, add that as an additional unique subsequence.

ending with '1' can have a '0' appended to make a new subsequence.

ending with '0' can have a '1' appended to make a new subsequence,

plus the new subsequence consisting of the single character '1'.

Initialize variables for dynamic programming.

let endingWithOneCount = 0;

const MODULUS = 1e9 + 7;

for (const char of binary) {

if (char === '0') {

// Return the final answer.

has_zero = False

mod = 10**9 + 7

else:

if has zero:

for char in binary:

if char == "0":

has_zero = True

return total_unique_subseq

assignment operations for each character.

return uniqueGoodSubsequencesCount;

} else {

let endingWithZeroCount = 0;

// ans will accumulate the final answer.

let uniqueGoodSubsequencesCount = 0;

// f represents the count of unique good subsequences ending with 1.

// a represents the count of unique good subsequences ending with 0.

// The modulus to ensure the answer stays within the integer limit.

// If the current character is '0', update the count of subsequences ending with 0.

// If the current character is '1', update the count of subsequences ending with 1.

// Add the counts of subsequences ending with '0' and '1', as well as the sequence "0" if present.

uniqueGoodSubsequencesCount = (uniqueGoodSubsequencesCount + endingWithOneCount + endingWithZeroCount) % MODULUS;

endingWithOneCount = (endingWithOneCount + endingWithZeroCount + 1) % MODULUS;

endingWithZeroCount = (endingWithZeroCount + endingWithOneCount) % MODULUS;

// The sequence "0" is always considered in the unique subsequences.

// As we found a '0', the flag is set to 1

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

if (binary.charAt(i) == '0') {

containsZero = 1;

} else {

Update count ones with count zeros since every subsequence

If we've encountered a zero, add that as an additional unique subsequence.

final int MODULO = (int) 1e9 + 7; // Defining the modulo value as a constant

endsWithZero = (endsWithZero + endsWithOne) % MODULO;

endsWithOne = (endsWithOne + endsWithZero + 1) % MODULO;

// Calculate the total by adding subsequences ending with '1', ending with '0'

int endsWithOne = 0; // Initialize variable to store subsequences that end with '1'.

int endsWithZero = 0; // Initialize variable to store subsequences that end with '0'.

int containsZero = 0; // A flag to indicate if there's at least one '0' in the sequence.

// If the current character is '0', update number of subseq. ending with '0'

// If the current character is '1', update number of subseq. ending with '1'

// Also, add the flag value to include the empty subsequence if there was at least one '0'

ending with '0' can have a '1' appended to make a new subsequence,

plus the new subsequence consisting of the single character '1'.

We've encountered a zero, so we note that.

'count zeros' stores the count of subsequences ending with '0'.

def numberOfUniqueGoodSubsequences(self, binary: str) -> int:

Modular constant to avoid overflow on large numbers.

Initialize variables for dynamic programming.

Iterate through the given binary string character by character. for char in binary: **if** char == "0": # Update count zeros with count ones since every subsequence # ending with '1' can have a '0' appended to make a new subsequence.

Aggregate total unique subsequences from those ending in '1' and '0'. total_unique_subseq = (count_ones + count_zeros) % mod

Java

class Solution {

```
int totalUniqueGoodSubsequences = (endsWithOne + endsWithZero + containsZero) % MODULO;
        return totalUniqueGoodSubsequences;
C++
class Solution {
public:
    int numberOfUniqueGoodSubsequences(string binary) {
        const int MOD = 1e9 + 7; // Define a constant for modulo to keep numbers within the integer range
        int endsWithOne = 0; // 'f' variable is now 'endsWithOne', tracks subsequences that end with '1'
        int endsWithZero = 0; // 'g' variable is now 'endsWithZero', tracks subsequences that end with '0'
        int hasZero = 0; // 'ans' variable is now 'hasZero', to indicate if there is at least one '0' in the string
        // Iterate through the given binary string
        for (char& c : binary) {
            if (c == '0') {
                // If current character is '0', update subsequences ending with '0'
                endsWithZero = (endsWithZero + endsWithOne) % MOD:
                // Record that the string contains at least one '0'
                hasZero = 1;
            } else {
                // If current character is '1', update subsequences ending with '1'
                // The current count is increased by the count of subsequences
                // ending in '0' plus this new '1' to form new unique subsequences
                endsWithOne = (endsWithOne + endsWithZero + 1) % MOD;
        // Calculate entire number of unique good subsequences
        // which is the sum of subsequences ending with '0', '1', plus '0'
        // if it exists in the string
        int totalUniqueGoodSubsequences = (hasZero + endsWithOne + endsWithZero) % MOD;
        return totalUniqueGoodSubsequences;
};
TypeScript
```

```
# 'count ones' stores the count of subsequences ending with '1'.
# 'count zeros' stores the count of subsequences ending with '0'.
# 'has zero' tracks if a subsequence with '0' has been observed.
count ones = 0
count zeros = 0
```

class Solution:

Time and Space Complexity **Time Complexity** The time complexity of the algorithm is O(N), where N is the length of the input string binary. This stems from the fact that the algorithm processes each character of the input string exactly once, performing a constant number of arithmetic operations and

Space Complexity

The space complexity of the algorithm is 0(1). There are only a few integer variables (f, g, ans, and mod) used to track the state throughout the processing of the input string, and their memory usage does not scale with the size of the input.