2450. Number of Distinct Binary Strings After Applying Operations

Medium Math String Leetcode Link

Problem Description

In this problem, we have a binary string `s`, meaning it contains only characters `0` and `1`. Along with this binary string, we're given a positive integer `k`. We're allowed to perform a specific operation on the string as many times as we wish, which is to choose any substring of length `k` and flip all the characters within it (i.e., turn `1`s into `0`s and `0`s into `1`s).

Our goal is to determine the number of distinct binary strings we can produce from string `s` by applying the said operation any number of times. Since this number can be large, our final answer must be presented modulo `10^9 + 7`.

In terms of definitions here, a **binary string** is simply a sequence of `0`s and `1`s. A **substring** refers to a sequence of characters that are consecutive within the larger string.

Intuition

Let's dive into the intuition behind the provided solution:

We are dealing with a binary string, and flipping operations always include substrings of the same size, `k`. Notice that once we pick a particular substring to flip, the result is independent of the characters outside of this substring. This means that each substring can be considered in isolation. Also notable is that flipping a substring twice will revert it to the original state. Hence, each unique substring of length `k` allows for 2 distinct outcomes: flipped and unflipped.

Now, consider the unique substrings of length `k` that can be obtained from `s`. If `s` has `n` characters, there are `n - k + 1` possible distinct substrings of length `k`. Each of these substrings can be in one of two states after a flip operation: their original state, or their flipped state. This leads us to the conclusion that there are at most `2^(n - k + 1)` distinct strings that can be obtained after performing any number of flip operations on the binary string `s`.

We say "at most" because if `k` is equal to the length of `s`, no matter how many times we operate, there will only be 2 distinct strings, the original and its complete flip. The formula accounts for this because when `k` equals the length of `s`, `n - k + 1` equals 1, indicating just two options.

The python solution appears to be a direct implementation of this intuition:

```python
def countDistinctStrings(self, s: str, k: int) -> int:
    return pow(2, len(s) - k + 1) % (10**9 + 7)
```

Thus, we use the `pow` function in Python to calculate `2^(n - k + 1)` and then take the modulo `%` with `10^9 + 7` to ensure the answer fits within the proper range.

This solution assumes that all substrings of length `k` within `s` can be independently flipped to create new combinations. The essence of the solution hinges upon the concept of binary choice for every substring of length `k`, which, when combined, leads to the exponential number of distinct strings attainable through the operation.

Solution Approach

The solution approach for this problem is straightforward and does not involve complex algorithms or data structures. In fact, it elegantly demonstrates the power of combinatorics in algorithmics.

Here is what the solution does step by step:

1. **Problem Analysis**: The first step is the recognition that each k-length substring has two states when flipped (original and flipped). Since we have `n - k + 1` potential unique substrings, we simply calculate the number of different combinations possible by using powers of 2.

2. **Mathematical Foundation**: The combinatorial principal applied here is represented by the expression `2^(n - k + 1)`. For each of the `n - k + 1` substrings, there are 2 possibilities after a flip operation. Thus, the total possibilities are the product of all these independent possibilities (2 multiplied by itself `n - k + 1` times).

3. **Implementation**: The Python `pow` function is used to calculate this power of 2. The `pow` function offers a built-in, computationally efficient way to calculate powers, even for large exponents. The expression is:

   ```python
   pow(2, len(s) - k + 1)
   ```

   This calculates `2` raised to the power of `len(s) - k + 1`.

4. **Modulo Operation**: Since the problem specifies that the output should be modulo `10^9 + 7`, we use the modulo operator `%`. In Python, this is as simple as appending `% (10**9 + 7)` to the `pow` function call. This operation ensures that the result is within the specified range, which is a common requirement to avoid overflow in problems dealing with combinatorics and large numbers.

The overall solution is concise because the problem does not require generation or enumeration of each distinct string. It just asks for the count, which allows for an elegant mathematical shortcut. The lack of Reference Solution Approach leaves the implementation part relatively straightforward given the direct translation of the math into code.

The final solution code is:

```python
class Solution:
    def countDistinctStrings(self, s: str, k: int) -> int:
        return pow(2, len(s) - k + 1, 10**9 + 7)
```

Notice the added third parameter to the `pow` function, which takes care of the modulo operation in a more efficient manner, potentially reducing the overhead of large number manipulation by performing the modulo at each step of power multiplication.

No additional data structures are used or needed, and no traditional algorithms are in play here; it's primarily a demonstration of understanding exponential growth and its representation in code.

Example Walkthrough

Let's go through an example to understand how the solution works.

Suppose our binary string `s` is `"10101"` and our integer `k` is `2`. This means we are looking at every possible unique substring of length `2` that can be flipped to get a new combination.

Original string `s`: `"10101"`

All unique substrings of length `2`: `"10"`, `"01"`, `"10"`, `"01"`

- For substring `"10"`, by flipping, we can get `"01"`.
- For substring `"01"`, by flipping, we can get `"10"`.

There are `4` unique substrings of length `2` since `n = 5` and `k = 2` (`n - k + 1 = 5 - 2 + 1 = 4`). For each of these substrings, we can have two states either original or flipped.

This corresponds to `2^4` combinations because we have 2 choices (flip or no flip) for each of the 4 places where a flip can occur.

Hence, according to our solution approach, we simply do:

```python
pow(2, 4) % (10**9 + 7)  # Using the corresponding values of n and k for this example
```

This evaluates to:

```python
16 % (10**9 + 7)  # Which equals 16 since 16 is much less than 10**9 + 7
```

So, there are `16` distinct binary strings that can be produced by flipping any of the substrings of length `k=2` from our original string `s="10101"`.

The final answer in this example would be `16`. This example illustrates how the provided solution calculates the number of distinct binary strings efficiently.

Python Solution

```python
class Solution:
    def countDistinctStrings(self, s: str, k: int) -> int:
        # Calculates the number of distinct substrings of length k
        # by computing 2 to the power of the difference between
        # the string's length and k, then adding 1. The result is
        # then modulo by 10^9 + 7 to ensure it does not exceed that value.

        # Calculate the total count of distinct possible strings
        total_count = pow(2, len(s) - k + 1)

        # Take modulo with 10^9 + 7 to keep the number within the integer limit
        mod_total_count = total_count % (10**9 + 7)

        return mod_total_count  # Return the result after modulo operation

```

Java Solution

```java
class Solution {

    // Create a constant for the modulo operation
    // to avoid counting numbers larger than 1e9 + 7
    public static final int MODULO = (int) 1e9 + 7;

    /**
     * Count the number of distinct substrings of length k that can be generated.
     *
     * @param str The input string to process.
     * @param k The length of substrings to account for.
     * @return The count of distinct substrings modulo 1e9 + 7.
     */
    public int countDistinctStrings(String str, int k) {
        // Initialize the answer to 1, considering an empty string to start with
        int distinctSubstringCount = 1;

        // Iterate over each possible starting position for substrings of length k
        for (int i = 0; i <= str.length() - k; ++i) {
            // Each character can either be included or not in each of the k length window,
            // effectively doubling the possibilities every iteration.
            // Take a modulo to keep the value within the bounds of MODULO.
            distinctSubstringCount = (distinctSubstringCount * 2) % MODULO;
        }

        // Return the number of distinct substrings
        return distinctSubstringCount;
    }
}

```

C++ Solution

```cpp
#include <string>

class Solution {
public:
    // Define the modulus for large number handling
    static constexpr int MOD = 1e9 + 7;

    // Function to count the number of distinct substrings of length k in string s
    int countDistinctStrings(std::string s, int k) {
        // Initialize answer to '1', as we start with a single-character string
        int distinctCount = 1;

        // Iterate over the string to consider all possible substrings of length k
        for (int i = 0; i <= s.size() - k; ++i) {
            // Double the count for each character considered, modulo MOD
            distinctCount = (distinctCount * 2) % MOD;
        }

        // Return the final count of distinct substrings
        return distinctCount;
    }
};

```

Typescript Solution

```typescript
// Define the modulus for large number handling
const MOD: number = 1e9 + 7;

// Function to count the number of distinct substrings of length k in string s
function countDistinctStrings(s: string, k: number): number {
    // Initialize the count of distinct substrings
    let distinctCount: number = 1;

    // Iterate over the string to consider all possible substrings of length k
    for (let i = 0; i <= s.length - k; i++) {
        // Double the count for each substring considered, modulo MOD
        distinctCount = (distinctCount * 2) % MOD;
    }

    // Return the final count of distinct substrings
    return distinctCount;
}

```

Time and Space Complexity

The time complexity of the function is `O(1)` because it consists of only one operation: calculating the power of 2 which is done in constant time regardless of the input size.

The space complexity of the function is also `O(1)` since the space used does not depend on the input size and only uses a fixed amount of space for intermediate calculations and to store the result.