481. Magical String

Medium | Two Pointers | String

Problem Description

The problem presents a unique sequence called a "magical string" that is built based on its own sequence of numbers. We start with a string `s` that only contains the characters `'1'` and `'2'`. The magic is that when we group the characters in `s` by the count of consecutive '1's and '2's, these grouped counts form the same string `s`.

As an example, we start with the known sequence: "1221121221221121122...". Grouping the consecutive characters yields "1 22 11 2 1 22 1 22 11 2 11 22 ...", and then the counts of '1's and '2's in these groups are "1 2 2 1 1 2 1 2 2 1 2 2 ...", which is the same as the original sequence `s`.

The task is to determine the count of '1's in the first `n` characters of the magical string `s`. Specifically, given an integer `n`, return how many '1's appear in the initial segment of length `n` in the string `s`.

Intuition

To solve this problem, we don't need to generate the entire string, which could be very long. Instead, we can build the string `s` only as far as necessary to count the number of '1's within the first `n` characters.

The solution uses a list `s` to simulate the magical string and a pointer `i` to keep track of where in `s` we are currently looking to determine how many times to repeat the next character. We begin with the known start of the magical string as [1, 2, 2].

The core idea is to iteratively extend the magical string based on its definition. For each step, we look at the value of `s[i]`, which tells us how many times to repeat the next character. We alternate the characters to add based on the last character in the string: if it's a '1', we add '2's; if it's a '2', we add '1's. The number of characters to add is equal to the current value of `s[i]`.

The use of `pre` helps us know what the last character was (either '1' or '2'), and `cur` is used to determine what the next character should be, by switching between '1' and '2'.

We continue extending the string until its length is at least `n`. Once the length of `s` reaches `n`, we simply take the first `n` elements of `s` and count the number of '1's to get our answer.

Solution Approach

The implementation relies on a simple Python list `s` to simulate the construction of the magical string. An integer `i` serves as a pointer that moves through the list, indicating how many times the next character should be repeated. Here's how the implementation unfolds:

1. We initialize the representation of the magical string as `s = [1, 2, 2]`. This is because the sequence always starts with "122".
2. We set the pointer `i = 2`. This is because `s[2]` points to the second '2' in the initial list, which indicates that the next sequence to be added should consist of two numbers. The pointer `i` will indicate which number in `s` we should look at to determine the next sequence to append to `s`.
3. We then enter a while loop which will run as long as the length of `s` is smaller than `n`, ensuring that we build enough of the magical string to count the number of '1's in the first `n` characters.
4. Inside the loop, we first store the last element of the current string in the variable `pre`. This is either a 1 or a 2, and it represents the last character in the string.
5. Next, we calculate `cur` as `3 - pre`. Since we only have '1' and '2' in our sequence, if the last character (pre) is '1', `cur` will be '2' (since `3 - 1 = 2`), and if it's '2', `cur` will be '1'.
6. The next step is to extend the list `s` by adding `cur` repeated `s[i]` times. This is akin to saying "if we have a '2' in `s[i]`, and `cur` is '1', then append '1' to `s` two times." We append `[cur] * s[i]` to `s`.
7. We then increment `i` by 1 to move the pointer to the next character in `s`.
8. Once we exit the while loop, it means we've built a section of the magical string that is at least as long as `n`. The last step is to return the count of '1's in the first `n` characters of `s` by using `s[:n].count(1)`.

This approach is efficient because it constructs only as much of the magical string as is necessary to determine the number of '1's within the first `n` elements. It uses simple list operations and arithmetic to achieve this task.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we are asked to find the number of '1's in the first 10 characters of the magical string `s`.

1. We start off by initializing our list `s` with the known beginning of the magical string: [1, 2, 2].
2. The pointer `i` is initially set to 2, since `s[2]` points to '2', which will determine what we append next to `s`.
3. Our `while` loop continues as long as the length of `s` is less than 10 (the `n` value in our example). At the start, the length of `s` is 3, so we enter the loop.
4. The last element, `pre`, of the current string `s` is 2.
5. We calculate `cur` as `3 - pre`. Since `pre` is 2, `cur` becomes 1.
6. We extend `s` by adding `cur` repeated `s[i]` times. In this case, since `s[i]` is 2 and `cur` is 1, we append two '1's to `s`, resulting in `s` becoming [1, 2, 2, 1, 1].
7. We then increment `i` by 1, moving the pointer to the next character in `s`, so `i` is now 3.
8. Now, `pre` is 1 (the last element in `s`), so now `cur` will be `3 - pre`, which is 2. `s[i]` is 1, so we append one '2' to `s`, changing `s` to [1, 2, 2, 1, 1, 2].
9. We repeat steps 4 through 7, with `i` now at 4. `s[i]` is 1, the current `pre` is 2, so `cur` is 1. We add one '1' to `s` to get [1, 2, 2, 1, 1, 2, 1].
10. Continuing this process, we increment `i`, calculate `cur`, and extend `s` until `s` is at least 10 characters long. After a few iterations, `s` is [1, 2, 2, 1, 1, 2, 1, 2, 2, 1], and the length of `s` is 10.
11. The loop stops since `s` now has a length of 10. We count the number of '1's in `s[:10]`, which is 5.

Thus, for `n = 10`, our function would return 5, as there are five '1's in the first 10 characters of the magical string `s`.

Solution Implementation

Python

```python
class Solution:
    def magicalString(self, n: int) -> int:
        # Initialize the magical string with the known starting sequence
        magical_str = [1, 2, 2]

        # Use index to track the position in the string for generating the next elements
        index = 2

        # Generate the magical string until its length is at least 'n'
        while len(magical_str) < n:
            # Get the last value in the magical string
            last_value = magical_str[-1]

            # The current value to be appended is the "opposite" of the last value (1 switches to 2, and 2 switches to 1)
            current_value = 3 - last_value

            # Append 'current_value' to the list as many times as the value at the current 'index'
            magical_str.extend([current_value] * magical_str[index])

            # Move to the next index
            index += 1

        # Slice the magical string until 'n' and count the occurrences of `1`
        return magical_str[:n].count(1)
```

Java

```java
class Solution {
    public int magicalString(int n) {
        // Initialize the magical string as a list with the first three numbers
        List<Integer> magicalStr = new ArrayList<>(Arrays.asList(1, 2, 2));

        // Use a pointer to iterate through the magical string to generate the next numbers
        int i = 2; // Starting from index 2 because the first three numbers are already in the list
        while (magicalStr.size() < n) {
            int lastNum = magicalStr.get(magicalStr.size() - 1); // Get the last number in the current magical string
            int nextNum = 3 - lastNum; // Calculate the next number (if lastNum is 1, then nextNum is 2; if lastNum is 2, then nextNu

            // Add 'nextNum' to the magical string 's.get(i)' times as per the current number's frequency
            for (int j = 0; j < magicalStr.get(i); ++j) {
                magicalStr.add(nextNum);
            }

            i++; // Move to the next number
        }

        // Count the number of occurrences of 1 in the first 'n' elements of the magical string
        int countOnes = 0;
        for (int idx = 0; idx < n; ++idx) {
            if (magicalStr.get(idx) == 1) {
                countOnes++;
            }
        }

        // Return the count of 1's in the first 'n' elements of the magical string
        return countOnes;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int magicalString(int n) {
        std::vector<int> magical_seq = {1, 2, 2}; // Initialize the magical sequence with its first three elements

        // Use 'index' to iterate through the magical sequence
        // The loop continues until the size of magical_seq is at least n
        for (int index = 2; magical_seq.size() < n; ++index) {
            int last_number = magical_seq.back(); // Get the last number in the current sequence
            int next_number = 3 - last_number; // Determine the next number to add, which will be 1 if the last is 2, and 2 if the la

            // Append the next number to the sequence s[i] times where s[i] is the current element at position i
            // This loop controls the count of the next number to be appended
            for (int count = 0; count < magical_seq[index]; ++count) {
                magical_seq.emplace_back(next_number); // Append the number to the end of the sequence
            }
        }

        // Count the number of 1's up to the nth element and return that count
        return std::count(magical_seq.begin(), magical_seq.begin() + n, 1);
    }
};
```

TypeScript

```typescript
function magicalString(n: number): number {
    // Initialize the magical string with the known beginning sequence
    let magicalStr = [...'1221121'];
    // Initialize the index for counting group occurrences
    let readIndex = 5;

    // Generate the magical string up to the required length 'n'
    while (magicalStr.length < n) {
        // Get the last character of the current magical string
        const lastChar = magicalStr[magicalStr.length - 1];
        // Append the opposite character to the string ('1' becomes '2', and '2' becomes '1')
        magicalStr.push(lastChar === '1' ? '2' : '1');
        // If the current read index character is '2', repeat the action once more
        if (magicalStr[readIndex] !== '1') {
            magicalStr.push(lastChar === '1' ? '2' : '1');
        }
        // Move to the next index
        readIndex++;
    }

    // Calculate the number of '1's in the first 'n' characters of the magical string
    return magicalStr.slice(0, n).reduce((count, char) => count + (char === '1' ? 1 : 0), 0);
}
```

```python
class Solution:
    def magicalString(self, n: int) -> int:
        # Initialize the magical string with the known starting sequence
        magical_str = [1, 2, 2]

        # Use index to track the position in the string for generating the next elements
        index = 2

        # Generate the magical string until its length is at least 'n'
        while len(magical_str) < n:
            # Get the last value in the magical string
            last_value = magical_str[-1]

            # The current value to be appended is the "opposite" of the last value (1 switches to 2, and 2 switches to 1)
            current_value = 3 - last_value

            # Append 'current_value' to the list as many times as the value at the current 'index'
            magical_str.extend([current_value] * magical_str[index])

            # Move to the next index
            index += 1

        # Slice the magical string until 'n' and count the occurrences of `1`
        return magical_str[:n].count(1)
```

Time and Space Complexity

Time Complexity

The time complexity of the function primarily comes from the while loop that generates the magical string until its length is at least `n`. In each iteration, the loop appends up to `s[i]` elements to the list `s` where `s[i]` could be either `1` or `2`. This results in a maximum of `2` additions per iteration. However, since each iteration of the loop adds at least one element, and we iterate until we have `n` elements, the overall time complexity can be approximated as `O(n)`.

Space Complexity

The space complexity of this function is also defined by the size of the list `s`, which grows to match the input `n` in the worst-case scenario. Since we need to store each element of the magical string up to the `n`th position, the space complexity is `O(n)`.