926. Flip String to Monotone Increasing

Medium String Dynamic Programming Leetcode Link

Problem Description

The problem presents us with a binary string `s` which consists only of the characters '0' and '1'. A monotone increasing binary string is defined as one where all '0's appear before any '1's. This means that there is no '1' that comes before a '0' in the string.

However, the input string `s` may not be monotone increasing already, so we need to transform it into one by flipping some of its characters. Flipping a character means changing a '0' to a '1' or a '1' to a '0'. The objective is to perform the minimum number of such flips so that the resulting string is monotone increasing.

We need to return the smallest number of flips necessary to convert the string `s` into a monotone increasing string.

Intuition

To solve this problem, we can use a dynamic programming approach to keep track of the minimum flips required to make prefixes and suffixes of the string monotone increasing. The core of the solution revolves around recognizing that for any position in the string, we can split the string into two parts:

- the prefix (all the characters before this position), which we want to turn into all '0's,
- and the suffix (all the characters from this position), which we want to turn into all '1's.

Now, if we know how many flips it would take to turn the prefix into all '0's and the suffix into all '1's, the total flips for the whole string would be the sum of both.

To compute these efficiently, we build two arrays, `left` and `right`.

- The `left` array at position `i` contains the number of flips to turn all characters from `0 to i-1` into '0's (note this includes the character at position `i-1`).
- The `right` array at position `i` contains the number of flips to turn all characters from `i to n-1` into '1's.

With these arrays, we iterate through each position in the string, representing all possible split points, compute the sum of flips from the `left` and `right` arrays for that position, and keep track of the minimum sum seen. After going through all the split points, the minimum sum found will be our answer, the least number of flips required to make the string monotone increasing.

Solution Approach

The solution uses dynamic programming to minimize the number of flips needed to make the string monotone increasing. Here's a step-by-step breakdown of how the provided code accomplishes this task:

1. Initialize two arrays, `left` and `right`, both of size `n+1`, where `n` is the length of the string `s`. These arrays will be used to store the cumulative number of '1's in the `left` array (from the start of the string to the current index) and the cumulative number of '0's in the `right` array (from the current index to the end of the string), respectively.
2. Populate the `left` array by iterating from `1` to `n+1` (inclusive). For each index `i` in `left`, increment the current value by 1 if the corresponding character in the string `s[i - 1]` is '1'. This gives us the number of '1's that would need to be flipped to '0's if the split is made after index `i-1`.
3. Populate the `right` array by iterating from `n-1` down to `0` (inclusive). For each index `i` in `right`, increment the current value by 1 if the corresponding character in the string `s[i]` is '0'. This gives us the number of '0's that would need to be flipped to '1's if the split is made before index `i`.
4. Now, we must determine the optimal split point. Set an initial `ans` value to a large number, representing the upper bound of the flips (the code uses `0x3F3F3F3F` as this value).
5. Iterate through each possible split point (from `0` to `n+1` inclusive). At each split point `i`, compute the total flips required by adding `left[i]` (flips required for the prefix to become all '0's) and `right[i]` (flips required for the suffix to become all '1's).
6. Update `ans` with the minimum between the current `ans` and the total flips for the current split position `i`.
7. After completing the iteration through all split points, `ans` will contain the minimum number of flips required to make the original string `s` monotone increasing.

The algorithm efficiently uses dynamic programming to compute the answer in `O(n)` time, where `n` is the length of the string. It avoids recalculation of the cumulative sums by storing them in the `left` and `right` arrays, which is a common technique used in dynamic programming to optimize for time complexity. The code does not use any complicated data structures, relying only on arrays for storage, which makes for an elegant and efficient solution.

Example Walkthrough

Let's consider an example to illustrate the solution approach. Assume we have the binary string `s = "00110"` which we want to convert into a monotone increasing string using the minimum number of flips.

Step-by-Step Process:

1. **Initialize Arrays**:
   - Our string `s` has `n = 5` characters, so we initialize `left` and `right` arrays of size `n+1`, which is `6` in this case.
2. **Populate the `left` Array**:
   - The `left` array tracks the cumulative number of flips to turn all characters to '0's. We iterate through the string and our `left` array will be built as follows:

```
Initialize left = [0, 0, 0, 0, 0, 0]
After processing '0': left = [0, 0, 0, 0, 0, 0]
After processing '0': left = [0, 0, 0, 0, 0, 0]
After processing '1': left = [0, 0, 1, 1, 1, 1]
After processing '1': left = [0, 0, 1, 2, 2, 2]
After processing '0': left = [0, 0, 1, 2, 2, 2]
```

3. **Populate the `right` Array**:
   - The `right` array tracks the cumulative number of flips to turn all characters to '1's. We iterate from the end of the string and form our `right` array as:

```
Initialize right = [0, 0, 0, 0, 0, 0]
After processing '0': right = [1, 1, 1, 1, 1, 0]
After processing '1': right = [1, 1, 1, 1, 0, 0]
After processing '1': right = [1, 1, 1, 0, 0, 0]
After processing '0': right = [1, 1, 0, 0, 0, 0]
After processing '0': right = [1, 0, 0, 0, 0, 0]
```

4. **Determine Optimal Split Point**:
   - We set our initial answer as `ans = Infinity` (or a very large value) for comparison purposes.
   - We then iterate through the possible split points to calculate the minimum flips:

```
At split 0: flips = left[0] + right[0] = 0 + 1 = 1
At split 1: flips = left[1] + right[1] = 0 + 0 = 0 (Minimum flips so far)
At split 2: flips = left[2] + right[2] = 0 + 0 = 0
At split 3: flips = left[3] + right[3] = 1 + 0 = 1
At split 4: flips = left[4] + right[4] = 2 + 0 = 2
At split 5: flips = left[5] + right[5] = 2 + 0 = 2
```

5. **Find Minimum Flips**:
   - The minimum number of flips required to make the string monotone increasing occurs at split positions 1 and 2, with `0` flips.
   - Thus, `ans = 0`.

Therefore, the given string `s = "00110"` is already a monotone increasing string and does not require any flips. Hence, the answer is `0`.

This example clearly demonstrates that by keeping track of the number of flips for making all characters to the left '0's and all characters to the right '1's for every position, we can easily compute the minimum total number of flips needed by considering all possible split points.

Python Solution

```python
class Solution:
    def minFlipsMonoIncr(self, s: str) -> int:
        # Calculate the length of the input string
        length = len(s)

        # Initialize the arrays to hold the number of 1s to the left (inclusive)
        # and the number of 0s to the right (inclusive) of each position
        ones_to_left = [0] * (length + 1)
        zeros_to_right = [0] * (length + 1)

        # Variable to hold the final minimum flips answer
        min_flips = float('inf')

        # Populate the ones_to_left array by counting the number of 1s to the left of each position
        for i in range(1, length + 1):
            ones_to_left[i] = ones_to_left[i - 1] + (1 if s[i - 1] == '1' else 0)

        # Populate the zeros_to_right array by counting the number of 0s to the right of each position
        for i in range(length - 1, -1, -1):
            zeros_to_right[i] = zeros_to_right[i + 1] + (1 if s[i] == '0' else 0)

        # Calculate the minimum flips required for a monotonically increasing string at each position
        # by adding the number of 1s to the left and 0s to the right for every possible split
        for i in range(0, length + 1):
            min_flips = min(min_flips, ones_to_left[i] + zeros_to_right[i])

        # Return the minimum number of flips required to make the string monotonically increasing
        return min_flips

```

Java Solution

```java
class Solution {

    /**
     * Calculate the minimum number of flips to make a binary string monotone increasing.
     *
     * @param s The input binary string.
     * @return The minimum number of flips.
     */
    public int minFlipsMonoIncr(String s) {
        int length = s.length(); // Length of the input string

        // Create arrays to store the prefix and suffix sums.
        int[] prefixOnes = new int[length + 1];
        int[] suffixZeros = new int[length + 1];

        // To hold the cumulative minimum number of flips.
        int minFlips = Integer.MAX_VALUE;

        // Calculate the prefix sums of 1s from the beginning of the string to the current position.
        for (int i = 1; i <= length; i++) {
            prefixOnes[i] = prefixOnes[i - 1] + (s.charAt(i - 1) == '1' ? 1 : 0);
        }

        // Calculate the suffix sums of 0s from the end of the string to the current position.
        for (int i = length - 1; i >= 0; i--) {
            suffixZeros[i] = suffixZeros[i + 1] + (s.charAt(i) == '0' ? 1 : 0);
        }

        // Iterate through all possible positions to split the string into two parts
        // and find the minimum number of flips by combining the count of 1s in the prefix
        // and the count of 0s in the suffix.
        for (int i = 0; i <= length; i++) {
            minFlips = Math.min(minFlips, prefixOnes[i] + suffixZeros[i]);
        }

        // Return the cumulative minimum number of flips.
        return minFlips;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function that returns the minimum number of flips to make the string monotonically increasing.
    int minFlipsMonoIncr(string s) {
        int size = s.size();
        vector<int> prefixOnes(size + 1, 0), suffixZeroes(size + 1, 0);
        int minFlips = INT_MAX;

        // Populate the prefixOnes to count the number of 1's from the start.
        for (int i = 1; i <= size; ++i) {
            prefixOnes[i] = prefixOnes[i - 1] + (s[i - 1] == '1');
        }

        // Populate the suffixZeroes to count the number of 0's from the end.
        for (int i = size - 1; i >= 0; --i) {
            suffixZeroes[i] = suffixZeroes[i + 1] + (s[i] == '0');
        }

        // For each position in the string, calculate the total flips
        // by adding the number of 1's in the prefix to the number of 0's in the suffix.
        // This sum represents the number of flips to make the string monotonically increasing
        // if the cut is made between the current position and the next.
        for (int i = 0; i <= size; i++) {
            // Find the minimum number of flips.
            minFlips = min(minFlips, prefixOnes[i] + suffixZeroes[i]);
        }

        return minFlips;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the minimum number of flips needed to make a binary string
 * increasing (each '0' should come before each '1').
 *
 * @param {string} s - The binary string to be processed.
 * @return {number} - The minimum number of flips required.
 */
const minFlipsMonoIncr = (s: string): number => {
    const n: number = s.length;
    let prefixSum: number[] = new Array(n + 1).fill(0);

    // Compute the prefix sum of the number of '1's in the string
    for (let i = 0; i < n; ++i) {
        prefixSum[i + 1] = prefixSum[i] + (s[i] === '1' ? 1 : 0);
    }

    let minFlips: number = prefixSum[n]; // Initialize minFlips with total number of '1's

    // Try flipping at each position, find the point that minimizes the
    // number of flips needed to make the string monotonically increasing
    for (let i = 0; i < n; ++i) {
        const flipsIfSplitHere: number =
            prefixSum[i] + // Number of '1's to flip to '0's before position i
            (n - i - (prefixSum[n] - prefixSum[i])); // Number of '0's to flip to '1's after position i, excluding i

        minFlips = Math.min(minFlips, flipsIfSplitHere);
    }

    return minFlips;
};

```

Time and Space Complexity

Time Complexity

The given code computes the minimum number of flips needed to make a binary string monotonically increasing using dynamic programming. The main operations are iterating through the string twice and computing the minimum in another pass. Let's break it down:

1. Initializing two arrays, `left` and `right`, each of size `n + 1`, where `n` is the length of the string `s`. This takes `O(1)` time as it's just initializing the lists with zeros.
2. A single for-loop to fill the `left` array. The loop runs `n` times (where `n` is the length of the string `s`), resulting in a complexity of `O(n)`.
3. Another single for-loop in reverse to fill the `right` array, which is also `O(n)`.
4. A final loop that goes from `0` to `n` to find the `ans` (minimum flips needed). This loop also runs `n + 1` times and the `min` operation inside it takes `O(1)` time. The total time complexity for this step is `O(n)`.

Therefore, the overall time complexity is the sum of the individual complexities: `O(n) + O(n) + O(n) = O(3n)`. Since constant factors are ignored in big O notation, the final time complexity simplifies to `O(n)`.

Space Complexity

The space complexity is the amount of additional memory space required to execute the code, which depends on the size of the input:

1. Two arrays `left` and `right` each of length `n + 1` are created. Hence, the space taken by these arrays is `2 * (n + 1)`.
2. Constant space is used for variables `n`, `ans`, and the loop indices, which is `O(1)`.

Thus, the total space complexity is `O(2n + 2)` which simplifies to `O(n)` as lower order terms and constant factors are dropped in big O notation.