1375. Number of Times Binary String Is Prefix-Aligned

Medium Array

Problem Description

In this LeetCode problem, we are given an initially zeroed binary string of length `n`, where the indexes are **1-indexed**. Throughout a sequence of steps defined by an integer array `flips`, we flip the bits of this binary string from `0` to `1`. The `i`th element of `flips` represents the bit position in the binary string that will be flipped during the `i`th step of the process.

A binary string is considered **prefix-aligned** after the `i`th step if all the bits from the beginning of the string to position `i` are set to `1`, while the rest of the string remains at `0`. The task is to calculate the total number of times the binary string is prefix-aligned during the entire flipping process.

For example:

```
- Initial binary string of length 5: 00000
- flips sequence: [3,2,4,1,5]

After each flip:
- Step 1, flip position 3: 00100 (Not prefix-aligned as the first bit is still 0)
- Step 2, flip position 2: 01100 (Not prefix-aligned as the first two bits are not all 1)
- Step 3, flip position 4: 01110 (Not prefix-aligned as the first four bits are not all 1)
- Step 4, flip position 1: 11110 (Not prefix-aligned as the first four bits are not all 1)
- Step 5, flip position 5: 11111 (Prefix-aligned as all the bits are 1)

Hence, the binary string is prefix-aligned 1 time during the flipping process.
```

Understanding the problem is crucial before attempting to create a solution.

Intuition

The intuition behind the solution is to keep track of the highest bit position (let's call it `mx`) that has been flipped at each step. For each step `i`, we compare `mx` with the current step index `i`: if `mx` is equal to `i`, it means that all bits up to the current step index have been flipped to `1`, and hence the string is prefix-aligned.

Here is how this intuition is applied to the solution:

1. Initialize the counter `ans` to keep track of the number of times the string is prefix-aligned, and `mx` to record the highest bit position flipped so far.
2. Iterate through each flip in the sequence, keeping track of the current step number `i` (starting from 1).
3. Update `mx` to be the maximum between its current value and the bit position `x` of the current flip.
4. If `mx` is equal to `i` after flipping the bit at position `x`, increment `ans` by 1 since the string is currently prefix-aligned.
5. Continue the loop until all flips are processed.
6. Return `ans`, the total count of prefix-aligned occurrences.

Solution Approach

The solution provided follows an iterative approach with two primary variables in play: `ans` and `mx`. The variable `ans` serves as a counter for the instances when the binary string is prefix-aligned, and `mx` keeps track of the maximum index that has been flipped so far.

1. Initialize `ans` to `0`. This variable will count the number of times the binary string is prefix-aligned during the flip sequence.
2. Initialize `mx` to `0`. This variable represents the maximum position of the flipped bit encountered up to the current step.

We use a for loop to iterate through each flip in the provided `flips` list. The loop uses the built-in `enumerate` function in Python to loop over `flips`, starting from index `1` because the problem is `1-indexed`.

3. During each iteration of the loop, we get two values: `i`, the step number starting from `1`, and `x`, the current position to flip.
4. For each flip, update `mx` to be the maximum between the current `mx` and the flip position `x`. This is achieved by the expression `mx = max(mx, x)`, which ensures `mx` always reflects the farthest position that was flipped till the current step.
5. The check `mx == i` will be `True` if all bits from the start to the current position `i` are `1` (thus, the string is prefix-aligned). If so, we increment `ans` by `1`. This leverages the fact that a binary string is prefix-aligned if the maximum flipped position at step `i` is equal to `i` itself. Any flip sequence that has the largest flip within the range `[1, i]` at `i` ensures that all preceding bits are already flipped to `1`.
6. Finally, the loop concludes once all the elements in `flips` have been iterated, and the `ans` value is returned. `ans` now contains the number of times the binary string was prefix-aligned during the process.

This algorithm doesn't use any complex data structures and requires no additional space besides the two variables `ans` and `mx`, making it space-efficient. The time complexity of the algorithm is O(n) where n is the number of flips since we are going through each flip exactly once. It efficiently solves the problem by keeping track of only the current state necessary to determine prefix alignment at each step without reconstructing the binary string at each instance.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have an initially zeroed binary string of length `4` and the following flips sequence: `[1,3,2,4]`.

- **Initial binary string:** 0000

We iterate through each flip, updating the maximum position `mx` flipped, and checking after each step if the string is prefix-aligned.

- **Step 1, flip position 1:** The binary string after the flip becomes `1000`. `mx` is updated to `1`. Since `mx` (1) equals the step number `i` (1), we increment `ans` by 1. The string is **prefix-aligned**.
  - New binary string: 1000
  - `mx` = 1
  - `ans` = 1
- **Step 2, flip position 3:** The binary string becomes `1010`. The new `mx` is `3`, which is now the highest position flipped so far. Since `mx` (3) does not equal the step number `i` (2), `ans` is not incremented. The string is **not prefix-aligned**.
  - New binary string: 1010
  - `mx` = 3
  - `ans` still = 1
- **Step 3, flip position 2:** After this flip, the binary string is `1110`, and `mx` remains `3`. As `mx` (3) is not equal to the step number `i` (3), `ans` remains the same. The string is **not prefix-aligned**.
  - New binary string: 1110
  - `mx` still = 3
  - `ans` still = 1
- **Step 4, flip position 4:** The final binary string is `1111`. `mx` is updated to `4`, which now equals the step number `i` (4). We increment `ans` by 1 again because the string is **prefix-aligned**.
  - New binary string: 1111
  - `mx` = 4
  - `ans` = 1 + 1 = 2

At the end of the flipping process, we have encountered 2 instances where the binary string is prefix-aligned, which means our answer `ans` is `2`.

Using the solution approach, we keep track of the maximum flipped position and check for prefix alignment in each step efficiently. The variables `ans` and `mx` enable us to do this without needing additional data structures or performing complex operations. This makes the algorithm space-efficient and straightforward to implement with a time complexity of O(n), where n is the number of flips.

Solution Implementation

Python

````python
# This class contains a method to determine the number of times all light bulbs are blue.
class Solution:
    def numTimesAllBlue(self, flips: List[int]) -> int:
        # Initialize the number of moments when all bulbs are blue and the current maximum flipped bulb position.
        moments_all_blue = 0
        max_flipped_position = 0

        # Loop through each flip by index and value.
        for moment, flip_position in enumerate(flips, 1):
            # Update the maximum flipped position if the current flip position is greater.
            max_flipped_position = max(max_flipped_position, flip_position)

            # If the maximum flipped position equals the current moment index, all the bulbs are blue.
            moments_all_blue += max_flipped_position == moment

        # Return the total count of moments when all the bulbs are blue.
        return moments_all_blue
```

In the code above:

- `flips`: The list of light bulb positions to flip during each moment, starting from 1.
- `moments_all_blue`: The count of times when all lights up to the current moment (included) turned blue.
- `max_flipped_position`: The maximum position (index) among the flipped light bulbs. If the maximum position we have flipped so
- The `enumerate` function is used to loop through each flip with its corresponding moment, beginning with 1. The `enumerate` fur

It's important to import the `List` typing from the `typing` module to ensure the type hint is recognized by the Python interpret

```python
from typing import List
````

Java

```java
class Solution {
    public int numTimesAllBlue(int[] flips) {
        int numMomentsAllBlue = 0; // This will store the number of moments when all bulbs are blue
        int maxTurnedOnBulb = 0; // This will keep track of the highest numbered bulb that has been turned on

        // Iterate through the flips array. Each flip represents turning on the bulb at that index.
        for (int moment = 1; moment <= flips.length; ++moment) {
            // Update the maxTurnedOnBulb with the maximum value between the current max and the bulb flipped at this moment
            maxTurnedOnBulb = Math.max(maxTurnedOnBulb, flips[moment - 1]);

            // If the maximum turned-on bulb number equals the current moment, increment the numMomentsAllBlue counter
            // It means all bulbs up to that point are blue
            if (maxTurnedOnBulb == moment) {
                ++numMomentsAllBlue;
            }
        }

        // Return the total number of moments when all flipped-on bulbs are blue
        return numMomentsAllBlue;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // For using the max function

class Solution {
public:
    // Function to count the number of moments when all bulbs are blue
    int numTimesAllBlue(vector<int>& flips) {
        int countBlueMoments = 0; // Initialize a counter for the blue moments
        int maxFlipped = 0; // This will keep track of the maximum bulb number flipped so far

        // Loop through each flip in the flips vector
        for (int i = 1; i <= flips.size(); ++i) {
            // Update the maximum flipped bulb number if the current flip is greater
            maxFlipped = max(maxFlipped, flips[i - 1]);

            // If the maximum flipped bulb number equals the number of flips so far,
            // it means all bulbs up to that point are on (and hence blue)
            countBlueMoments += (maxFlipped == i) ? 1 : 0; // Use the ternary operator for the condition
        }

        // Return the total count of moments when all turned-on bulbs are blue
        return countBlueMoments;
    }
};
```

TypeScript

```typescript
function numTimesAllBlue(lightSwitches: number[]): number {
    let countAllBlue = 0; // This will hold the number of times all lights turned blue
    let maxSwitchedOn = 0; // Tracks the maximum light switch number turned on so far

    // Loop through each light switch flip
    for (let i = 1; i <= lightSwitches.length; ++i) {
        // Update maxSwitchedOn to be the highest of the current max or the current light switch flipped
        maxSwitchedOn = Math.max(maxSwitchedOn, lightSwitches[i - 1]);
        // If maxSwitchedOn is equal to the number of flips so far, all lights are blue
        if (maxSwitchedOn === i) {
            countAllBlue += 1;
        }
    }

    // Return the total number of times all lights have turned blue
    return countAllBlue;
}
```

````python
# This class contains a method to determine the number of times all light bulbs are blue.
class Solution:
    def numTimesAllBlue(self, flips: List[int]) -> int:
        # Initialize the number of moments when all bulbs are blue and the current maximum flipped bulb position.
        moments_all_blue = 0
        max_flipped_position = 0

        # Loop through each flip by index and value.
        for moment, flip_position in enumerate(flips, 1):
            # Update the maximum flipped position if the current flip position is greater.
            max_flipped_position = max(max_flipped_position, flip_position)

            # If the maximum flipped position equals the current moment index, all the bulbs are blue.
            moments_all_blue += max_flipped_position == moment

        # Return the total count of moments when all the bulbs are blue.
        return moments_all_blue
```

In the code above:

- `flips`: The list of light bulb positions to flip during each moment, starting from 1.
- `moments_all_blue`: The count of times when all lights up to the current moment (included) turned blue.
- `max_flipped_position`: The maximum position (index) among the flipped light bulbs. If the maximum position we have flipped so far (
- The `enumerate` function is used to loop through each flip with its corresponding moment, beginning with 1. The `enumerate` functio

It's important to import the `List` typing from the `typing` module to ensure the type hint is recognized by the Python interpreter:

```python
from typing import List
````

Time and Space Complexity

Time Complexity

The given algorithm consists of a single `for` loop that iterates through every element in the `flips` list exactly once. The enumeration of `flips` does not change the overall time complexity. Inside the loop, the algorithm performs a constant amount of work on each iteration, including a comparison and an assignment. Therefore, the time complexity of this function is `O(n)` where `n` is the number of elements in `flips`.

Space Complexity

The space complexity of the algorithm is defined by the amount of additional memory used by the algorithm as a function of the input size. Within the provided algorithm, `ans` and `mx` are the only variables that occupy extra space, and their memory footprint does not depend on the size of the input. As a result, the space complexity is `O(1)` because the amount of extra memory used is constant, irrespective of the input size.