1980. Find Unique Binary String

Medium Array String Backtracking Leetcode Link

Problem Description

The task is to find a binary string of length `n` that is not included in the given array `nums`, which contains `n` unique binary strings of the same length. The binary string to be found should consist only of '0's and '1's and should not match any of the binary strings in the given list. We are also given the freedom to provide any valid string that satisfies the condition in case multiple correct answers exist.

Intuition

The intuition behind this solution involves bit manipulation. Since there are `n` unique binary strings, and each is `n` bits long, there's a possibility that all binary representations from 0 to `n-1` are present. However, the pigeonhole principle states that since there are `n+1` possible binary numbers from 0 to `n`, there must be at least one number not represented by the `n` binary strings.

The solution uses a `mask` variable to store which of the binary numbers represented by the count of '1's in each string are already present in the given `nums` list. A loop goes through the strings and sets a bit in the `mask` to 1 for each count of '1's that occurs.

After that, we loop over the possible counts of '1's from 0 to `n` and look for a bit in the `mask` that is still 0 (which means this count of '1's has not been used). When we find such a bit, we construct a binary string with this count of '1's followed by the necessary count of '0's to make the length `n` and return it. This string is guaranteed not to be in the list because we generated it based on a count of '1's not present in the original list.

For example, if `n` is 3, and `nums` has ["000", "011", "101"], the counts of '1's are 0, 2, and 2 respectively. So, `mask` would have bits 0 and 2 set to 1 (00101 in binary). The number of '1's that is not represented is 1 and 3, and thus, the strings "010" or "111" would be valid outputs.

Solution Approach

The code uses the concept of a bit mask and bit manipulation to keep track of the counts of '1's that are present in the given binary strings `nums`. The algorithm proceeds as follows:

1. Initialize a variable `mask` to 0. This mask will have its bits set to 1 at positions that correspond to the counts of '1's found in the binary strings from `nums`.

2. For each binary string `x` in `nums`, convert the count of '1's into a position in the `mask` and set that position to 1. This is done by the operation `mask |= 1 << x.count("1")`. In essence, for a string `x`, if it contains 'k' number of '1's, the k-th bit of `mask` is set to 1.

3. Obtain `n`, which is the length of the input list `nums` (also the length of every binary string in `nums`).

4. Iterate from `i = 0` to `n` (inclusive) and check if the bit at the i-th position of `mask` is not set (which means that no string in `nums` has exactly `i` number of '1's). This is performed by checking if `mask >> i & 1 ^ 1` is `True`, where `mask >> i` shifts the `mask` `i` times to the right, creating a new mask that has the i-th bit of the original `mask` at the 0-th position. To check if the bit is not set, we use the XOR operation with 1 (`& 1 ^ 1`) which will yield `True` if and only if the bit was 0.

5. When an unset bit is found (`mask >> i & 1 ^ 1` is `True`), construct the binary string result by concatenating `i` number of '1's with `n-i` number of '0's to make the string's length equal to `n`. This resultant string is guaranteed to be unique and not to appear in `nums`.

6. Return the constructed string.

For example, if `n` is 3, and `nums` contains the strings ["000", "011", "101"], the `mask` after step 2 would be `5` (in decimal) or `101` in binary as the 0-th and 2-nd bits are set to 1. Looking for an unset bit in `mask`, we find that the 1-st and 3-rd positions are 0. Therefore, we can construct the strings "010" or "111" as our output, both of which have a length of 3 and do not appear in `nums`.

Example Walkthrough

Let's take an example with `n = 2` and `nums` being `["00", "01"]`. We want to find a binary string of length 2 that is neither "00" nor "01".

1. Initialize `mask` to 0. This will be used to track the count of '1's present in the binary strings from `nums`.

2. Iterate through each string in `nums`:
   - For "00", there are 0 '1's. We set the 0-th bit of `mask` using `mask |= 1 << 0`. Now `mask` is `01` in binary (1 in decimal).
   - For "01", there is 1 '1'. We set the 1-st bit of `mask` using `mask |= 1 << 1`. Now `mask` changes from `01` to `11` in binary (3 in decimal), since both 0-th and 1-st bits are set.

3. Now that `n` is 2, we check for bits in `mask` which are not set.

4. We start iterating from `i = 0` to `n` (until 2, inclusive):
   - For `i = 0`: Check `mask >> i & 1 ^ 1`. We have `mask >> 0` which is `11` (3) and `& 1` which gives the last bit '1', then `^ 1` would give 0 (False). The 0-th bit is set, so we continue.
   - For `i = 1`: Check `mask >> i & 1 ^ 1`. We have `mask >> 1` which is `1` (1) and `& 1` which gives '1', then `^ 1` would give 0 (False). The 1-st bit is also set, so we continue.
   - For `i = 2`: Check `mask >> i & 1 ^ 1`. We have `mask >> 2` which is `0` (0) and `& 1` which gives '0', then `^ 1` would give 1 (True). The 2-nd bit is not set; this means no string in `nums` has 2 '1's.

5. As soon as we find an i-th bit not set, we construct the binary string with `i` number of '1's followed by `n-i` number of '0's. Since `n=2` and `i=2`, we get "11" as the result. This string has exactly 2 '1's, and its length matches `n`.

6. Thus, the string "11" is returned. This string is of length `n`, not present in `nums`, and follows the correct format.

Python Solution

```python
class Solution:
    def findDifferentBinaryString(self, nums: List[str]) -> str:
        # Initialize a bitmask to keep track of the count of "1"s in the binary strings
        bitmask = 0

        # Iterate over each binary string in the nums list
        for binary_string in nums:
            # Count the number of "1"s and set the corresponding bit in the bitmask
            # The bit position is the count of "1"s
            bitmask |= 1 << binary_string.count("1")

        # Get the length of binary strings in the nums list (assuming they all have the same length)
        n = len(nums)

        # Find the smallest binary string with a different count of "1"s than those in nums
        for i in range(n + 1):
            # Check if there is no binary string in nums with i "1"s
            if bitmask >> i & 1 ^ 1:
                # Return the binary string with i "1"s followed by (n - i) "0"s
                # Ensuring it has the same length as other strings in nums
                return "1" * i + "0" * (n - i)

```

Java Solution

```java
class Solution {
    public String findDifferentBinaryString(String[] nums) {
        // Initialize a mask to keep track of the counts of 1s found in the binary strings.
        int mask = 0;

        // Iterate over each binary string in the input array.
        for (String binaryString : nums) {
            // Initialize a count for the number of '1's in the current string.
            int countOnes = 0;

            // Iterate through each character in the binary string.
            for (int i = 0; i < binaryString.length(); i++) {
                // If the character is '1', increment the counter.
                if (binaryString.charAt(i) == '1') {
                    countOnes++;
                }
            }

            // Set the corresponding bit in the mask for the count of '1's found.
            // This will help us to keep track of which counts are already present.
            mask |= 1 << countOnes;
        }

        // Start generating different binary strings to find one not in the input.
        for (int i = 0; ; i++) { // Infinite loop, will break when found a non-existing binary string.
            // Check if the bit representing the count of '1's at index 'i' is not set.
            if ((mask >> i & 1) == 0) {
                // Create a binary string with 'i' number of '1's followed by enough '0's to make it the same length as input strings
                // Then return the newly created binary string.
                String ones = "1".repeat(i);
                String zeros = "0".repeat(nums.length - i);
                return ones + zeros;
            }
        }
    }
}

```

C++ Solution

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

class Solution {
public:
    std::string findDifferentBinaryString(std::vector<std::string>& nums) {
        // Initialize a variable to serve as a bitmask where each bit represents
        // the count of '1's in the binary strings seen so far.
        int bitmask = 0;

        // Loop through the binary strings.
        for (auto& str : nums) {
            // Count the number of '1's in the current string.
            int countOnes = std::count(str.begin(), str.end(), '1');
            // Set the corresponding bit in the bitmask.
            bitmask |= 1 << countOnes;
        }

        // Loop indefinitely to find a binary string with a different count of '1's.
        for (int i = 0;; ++i) {
            // Check if the current count of '1's is not represented in the bitmask.
            // The expression (bitmask >> i) shifts the bitmask to the right by 'i' bits,
            // and then checks if the least significant bit is not set.
            if (((bitmask >> i) & 1) == 0) {
                // If not set, we found our number. Return a binary string with 'i' ones
                // followed by enough zeros to match the size of the input binary strings.
                return std::string(i, '1') + std::string(nums.size() - i, '0');
            }
        }
        // No return is needed here as the loop is guaranteed to return a string
        // because there are 2^N possible binary strings of length N, and only N of them
        // have unique counts of '1's, leaving at least one string that is different.
    }
};

```

Typescript Solution

```typescript
function findDifferentBinaryString(nums: string[]): string {
    // Initialize a variable to act as a bitmask where each bit position represents
    // the count of '1's in the binary strings seen so far.
    let bitmask: number = 0;

    // Loop through the binary strings.
    for (let str of nums) {
        // Count the number of '1's in the current string.
        let countOnes: number = [...str].filter(c => c === '1').length;
        // Set the corresponding bit in the bitmask.
        bitmask |= 1 << countOnes;
    }

    // Start an infinite loop to find a binary string with a different count of '1's.
    for (let i = 0; ; ++i) {
        // Check if the current count of '1's is not yet represented in the bitmask.
        // The expression (bitmask >> i) shifts the bitmask to the right by 'i' bits,
        // and then checks if the least significant bit is not set.
        if (((bitmask >> i) & 1) === 0) {
            // If not set, we have found our number. Return a binary string with 'i' '1's
            // followed by enough '0's to match the size of the input binary strings.
            return '1'.repeat(i) + '0'.repeat(nums[0].length - i);
        }
    }
    // No explicit return is necessary here as the loop condition assures a return,
    // because there are 2^N possible binary strings of length N, and only N of them
    // can have unique counts of '1's, ensuring at least one string that is different.
}

// The method names are not modified and the above function can be used directly.
// Usage example:
// let result = findDifferentBinaryString(["01", "10"]);
// console.log(result); // Outputs a binary string that is different from the input strings

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by iterating over the input list `nums` once, and then iterating through a range that is at most `n + 1`, where `n` is the length of `nums`.

- The first `for` loop iterates over each string in `nums` and involves a bitwise OR operation as well as counting the number of '1's in each string. Both of these operations occur in constant time (since the strings are of length `n` and binary string length is fixed in this scenario). Hence, this part of the loop runs in `O(n)` time.

- The second `for` loop runs from 0 to `n`, and each iteration involves a constant-time operation: a bitwise right shift followed by a bitwise AND, and a bitwise XOR. Therefore, the loop runs in `O(n)` time.

Combining both parts, the overall time complexity is `O(n + n)` which simplifies to `O(n)`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm beyond the input itself.

- The variable `mask` is an integer that requires `O(1)` space.

- No other additional data structures that grow with the size of the input are used in the algorithm.

Hence, the overall space complexity of the code is `O(1)`.