1888. Minimum Number of Flips to Make the Binary String Alternating

Problem Description

The given LeetCode problem involves operations on a binary string, which is a string composed solely of '0's and '1's. The objective is to make the string "alternating", which means no two consecutive characters in the string are the same. There are two types of operations that can be used to achieve this:

- **Type-1**: Remove the first character of the string and append it to the end. This is effectively a cyclic permutation of the string.
- **Type-2**: Flip a character in the string from '0' to '1' or from '1' to '0'.

The challenge is to determine the minimum number of Type-2 operations (flips) required to make the string alternating. We don't need to worry about the number of Type-1 operations since the question only asks for the count of Type-2 operations.

Intuition

The intuition behind the solution comes from observing patterns within an alternating string. Consider the two alternating patterns for a binary string: "010101..." and "101010...". Any valid alternating binary string can be transformed into either of these two patterns with Type-2 operations.

The key insight is to simulate the operations to move towards one of these patterns. Here's how the solution approach works:

1. Calculate the number of flips required to convert the current string into the "010101..." pattern. This is done by comparing each character in the binary string to the expected character in the "01" pattern. Keep count of mismatches.
2. Calculate the number of flips required for the "101010..." pattern. This is simply the length of the string minus the number of flips for "010101..." because every place that matches one pattern mismatches the other and vice versa.
3. Consider the effect of Type-1 operations. Since Type-1 operations are just cyclic permutations, simulate this by moving through the string and adjusting the mismatch count as if the first character is moved to the end, one by one.
4. At each step, recalculate the number of flips needed, taking into account the updated first character. Repeat this for the entire string length and track the minimum number of flips found.
5. The outcome is the minimum of these recalculated flip counts.

This algorithm works because it efficiently combines the permissibility of Type-1 operations to rearrange the string (without actually doing it) with the direct calculation of Type-2 operations required to achieve the pattern.

Solution Approach

The approach to solving this problem is as follows:

1. First, we need initial counts of mismatches for both alternating patterns for the input string `s`. This can be achieved by iterating over the string and checking each character if it matches the expected character in the pattern "010101...".
2. To implement this, we compare each character in `s` with the expected character in the pattern. For example, `c` should match `target[i & 1]`, where `i & 1` is the bitwise AND operator, used here to alternate between `0` and `1` as we move along the string.
3. We keep a running count `cnt` of how many characters do not match the expected "01" pattern. Since for an alternating string of length `n`, the other pattern is just the inverse, the count for the other pattern is `n - cnt`.
4. We then initialize the answer `ans` to be the minimum of `cnt` and `n - cnt` because we have the choice to convert to either pattern.
5. Next, we simulate Type-1 operations without actually permuting the string. We iterate over the string once again and, for each character, update the mismatch count `cnt` for the two patterns by considering that the first character is moved to the end. Specifically, we will reduce `cnt` by one if the first character differs from the expected character in the current position and adjust `cnt` considering it's now at the end of the string. This effectively simulates a cyclic shift to the left by one position.
6. After each simulated cyclic shift, we update `ans` with the minimum of the current `ans`, current `cnt`, and `n - cnt`, which keeps track of the minimum number of flips required after each simulated Type-1 operation.
7. After the loop, `ans` contains the minimum number of flips required to make the string alternating.

In terms of algorithms and data structures, this solution employs a greedy counting approach, where we greedily count mismatches against an expected pattern and use properties of bitwise operations to efficiently alternate between '0' and '1'. The solution also leverages in-place updates to minimize space complexity; no additional data structures are needed beyond simple variables for keeping track of counts and the final answer.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose the input binary string is `s = "0011"`. We are to find out the minimum number of flips (Type-2 operations) to make the string alternating ("0101" or "1010").

1. We start by calculating the number of flips needed for the pattern starting with `0` ("0101"). As we scan `s`, we notice:
   - `s[0]` = '0', no flip needed (matches "0101").
   - `s[1]` = '0', flip needed (does not match "0101", should be '1').
   - `s[2]` = '1', no flip needed (matches "0101").
   - `s[3]` = '1', flip needed (does not match "0101", should be '0').
   Therefore, we have 2 mismatches for the "0101" pattern, which means `cnt = 2`.
2. For the pattern starting with `1` ("1010"), we find that we also have 2 mismatches, because it is just the inverse of the other pattern, so `n - cnt = 2`.
3. We initialize our answer `ans = min(cnt, n - cnt)`, which in this case is `min(2, 2) = 2`.
4. Now, we simulate the Type-1 operations. We cyclically move the first character to the end and update the flip count. After the first shift, our string becomes "0110":
   - `s[1]` was '0', and now it's in the second position where it should be '1', so `cnt` increases to 3.
   - `s[0]` was '0', and now it's in the last position where it should be '0', so no change in `cnt`.
5. We then update `ans` with `min(ans, cnt, n - cnt)`. Since `cnt` increased, we have `min(2, 3, 2)`, so `ans` remains `2`.
6. We continue the simulation for two more shifts:
   - After second shift to "1100", `cnt` would decrease since the first '1' at the third position matches, and the last '0' still matches. So, `cnt` becomes 2.
   - After third shift to "1001", `cnt` would remain the same since the first '1' at the fourth position still matches, and the last '1' doesn't change the pattern.
7. At every step, we keep checking and updating the minimum flips needed, which in our example remains 2.

The final answer is 2 flips required to make the string alternating, which is the minimum number of flips found during the simulation.

Solution Implementation

Python

```python
class Solution:
    def minFlips(self, s: str) -> int:
        # Determine the length of the input string "s"
        n = len(s)

        # This string will be used to check against the input string "s"
        target = "01"

        # Count the number of flips required to match the "01" pattern starting with "0"
        initial_flips = sum(char != target[i % 2] for i, char in enumerate(s))

        # Initialize the minimum flips answer with the minimum
        # matching "01" or "10" pattern starting with "0"
        min_flips = min(initial_flips, n - initial_flips)

        # Loop over the string "s" to consider all cyclic permutations of "s"
        for i in range(n):
            # On each iteration, consider that the first character has moved to the end
            # Adjust the flip count accordingly
            initial_flips -= s[i] != target[i % 2]
            initial_flips += s[i] != target[(i + n) % 2]

            # Update the minimum flips answer considering the current cyclic permutation
            min_flips = min(min_flips, initial_flips, n - initial_flips)

        # Return the minimum flips answer found
        return min_flips
```

Java

```java
class Solution {

    // Method to find the minimum number of flips needed to make the string alternating.
    public int minFlips(String s) {
        // Length of the input string
        int stringLength = s.length();

        // String to represent the pattern "01" which we want to alternate in the final string
        String alternatingPattern = "01";

        // Count the number of mismatches between the input string and the pattern
        int mismatches = 0;
        for (int i = 0; i < stringLength; ++i) {
            if (s.charAt(i) != alternatingPattern.charAt(i % 2)) {
                ++mismatches;
            }
        }

        // Calculate the minimal flips needed as the minimum between mismatches
        // and the complement to the length of the string (because we can flip either "0"s or "1"s)
        int minFlips = Math.min(mismatches, stringLength - mismatches);

        // This for loop mimics the idea of a circular array by going once more through the string,
        // and adjusting mismatch count as if the string was rotated.
        for (int i = 0; i < stringLength; ++i) {
            // If the current character does not match the alternating pattern,
            // we remove one from mismatches assuming that character is moved to the back
            if (s.charAt(i) != alternatingPattern.charAt(i % 2)) {
                --mismatches;
            }

            // As the character "moves" to the end of the string, we need to check it against
            // the complementing pattern index.
            if (s.charAt(i) != alternatingPattern.charAt((i + stringLength) % 2)) {
                ++mismatches;
            }

            // Re-calculate the minimum flips after each rotation.
            minFlips = Math.min(minFlips, Math.min(mismatches, stringLength - mismatches));
        }

        // After going through all possible rotations, return the minimum flips found.
        return minFlips;
    }
}
```

C++

```cpp
class Solution {
public:
    int minFlips(string s) {
        int length = s.size(); // Get the size of the input string
        string target = "01"; // Define the target pattern
        int flips = 0; // Initialize the count of flips needed to transform the current string

        // Count how many flips are needed to match the string with the alternating "01" pattern
        for (int i = 0; i < length; ++i) {
            if (s[i] != target[i % 2]) { // Check each character against the target pattern
                ++flips;
            }
        }

        // Initialize the answer with the minimum of flips and its inverse
        // since the string can start with either '0' or '1'
        int minFlips = std::min(flips, length - flips);

        // Loop through the string considering that the string is circular for optimization
        for (int i = 0; i < length; ++i) {
            // If we remove a character that needs flipping from the start, decrease the flip count
            if (s[i] != target[i % 2]) {
                --flips;
            }
            // If we pretend the removed character is added to the end, and it needs flipping,
            // increase the flip count
            if (s[i] != target[(i + length) % 2]) {
                ++flips;
            }
            // Update the answer with the fewest flips required considering the circular rotation
            minFlips = std::min(minFlips, flips, length - flips));
        }

        // Return the minimum number of flips to make the string alternating
        return minFlips;
    }
};
```

TypeScript

```typescript
function minFlips(s: string): number {
    const lengthOfString: number = s.length;
    let alternatingPattern: string = '01'; // The desired pattern is an alternating sequence of '0' and '1'
    let flipCount: number = 0; // This will count the number of flips to convert the string to the desired pattern

    // Initial pass to count the number of flips needed if we consider the string as is
    for (let index = 0; index < lengthOfString; ++index) {
        if (s[index] !== alternatingPattern[index % 2]) {// '%' operator is used to alternate between '0' and '1'
            ++flipCount;
        }
    }

    // The number of flips should be the minimum between flipCount and the count in the opposite sequence
    let minimumFlips: number = Math.min(flipCount, lengthOfString - flipCount);

    // This loop considers the string as if it were circular, thus we repeat the initial process after virtually 'rotating' the s
    for (let index = 0; index < lengthOfString; ++index) {
        // Decrease the flip count when current character is the same as its position in the alternating pattern
        if (s[index] !== alternatingPattern[index % 2]) {
            --flipCount;
        }
        // Increase the flip count if the character matches the opposite pattern when considered in a circular rotation
        if (s[index] !== alternatingPattern[(index + lengthOfString) % 2]) {
            ++flipCount;
        }

        // Update the minimum flips considering the current rotation
        minimumFlips = Math.min(minimumFlips, flipCount, lengthOfString - flipCount);
    }

    return minimumFlips; // Return the minimum number of flips after considering all rotations
}
```

```python
class Solution:
    def minFlips(self, s: str) -> int:
        # Determine the length of the input string "s"
        n = len(s)

        # This string will be used to check against the input string "s"
        target = "01"

        # Count the number of flips required to match the "01" pattern starting with "0"
        initial_flips = sum(char != target[i % 2] for i, char in enumerate(s))

        # Initialize the minimum flips answer with the minimum between
        # matching "01" or "10" pattern starting with "0"
        min_flips = min(initial_flips, n - initial_flips)

        # Loop over the string "s" to consider all cyclic permutations of "s"
        for i in range(n):
            # On each iteration, consider that the first character has moved to the end
            # Adjust the flip count accordingly
            initial_flips -= s[i] != target[i % 2]
            initial_flips += s[i] != target[(i + n) % 2]

            # Update the minimum flips answer considering the current cyclic permutation
            min_flips = min(min_flips, initial_flips, n - initial_flips)

        # Return the minimum flips answer found
        return min_flips
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`. This is because there is a single loop traversing the string of length `n`. Inside the loop, only constant-time operations are performed, such as updating the `cnt` variable and comparing values. There are no nested loops or recursive calls that would increase the complexity.

The space complexity of the provided code is `O(1)`. The solution is using only a fixed number of variables (`n`, `cnt`, `ans`, `target`, and a couple of iterators in the loop) which does not grow with the size of the input. No additional data structures that scale with the input size are used.