1758. Minimum Changes To Make Alternating Binary String

Easy String

Problem Description

You are given a binary string `s` that consists only of characters '0's and '1's. An operation is defined as changing a single character from '0' to '1' or from '1' to '0'. The goal is to transform the string into an alternating string, meaning no two adjacent characters will be the same. For instance, '010101' and '101010' are examples of valid alternating strings. You are asked to find the minimum number of operations required to achieve an alternating string from the given string `s`.

Intuition

To arrive at the least number of operations to make the string alternating, we can consider two cases:

1. The string starts with '0': This means the string should follow the pattern '010101...'
2. The string starts with '1': This would make the string follow '101010...'

For any given string `s`, these two cases are the only possible alternating strings that can be made without considering shifting the whole string.

Since we only need to consider the number of mismatches, we can iterate through the string once and count how many characters do not match the alternating pattern for each of the cases. The operation `c != '01'[i & 1]` checks if the character `c` at index `i` does not match the expected alternating character ('0' if `i` is even, '1' if odd). For each character, `cnt` is incremented if it mismatches.

Once the number of mismatches (`cnt`) for one pattern is determined, the minimum number of operations needed can either be `cnt` (changing all mismatched characters to match the pattern) or `len(s) - cnt` (changing all matched characters to flip the pattern). The minimum of these two values is the fewest number of operations needed to make the string `s` alternating, as changing all mismatched or all matched characters will achieve the same result.

Thus, our approach to solving the problem is to:

- Count the number of mismatches against one of the possible patterns.
- Calculate the operations needed to correct mismatches and to flip matches.
- Return the lower value between the two operations calculated.

Solution Approach

The implementation provided in the reference solution applies a simple but efficient approach to solve the problem. It relies on string iteration, bitwise operations, and elementary arithmetic. No complex data structures are required. Here's a step-by-step breakdown of the approach used:

- The `enumerate` function is used to iterate over the input string `s`, providing both the index (`i`) and the character (`c`) for each iteration.
- For every character in the string, a check is performed using a bitwise AND operation `i & 1` which efficiently determines the parity of the index `i` (even or odd). If `i` is even, `i & 1` will be `0`; if odd, it will be `1`.
- Based on the index parity, the character is compared against the expected character in an alternating string starting with '0' ('0' for even indices, '1' for odd indices). This is done using `'01'[i & 1]`, which indexes into the string `'01'` to pick the expected character.
- The comparison `c != '01'[i & 1]` results in a boolean value, which is then automatically cast to an integer (`0` for `False`, `1` for `True`) when used in arithmetic operations such as summation.
- The `sum` function totals the number of mismatches `cnt` by counting every time the character does not meet the alternating condition based on its position (index parity).
- Finally, the solution calculates the number of operations to transform the input string into an alternating string by taking the minimum between `cnt` and `len(s) - cnt`. The latter represents the number of operations needed to achieve the opposite alternating pattern (starting with '1' instead of '0') which might require fewer operations.

The solution uses a common algorithmic pattern for problems of this nature: the greedy approach. It relies on the notion that a local optimal solution will lead to a globally optimal solution. In this case, addressing each character individually and deciding whether to flip it based on its position leads us to the minimum number of operations for the entire string.

Example Walkthrough

Let's consider a short example to illustrate the solution approach. Suppose we have the binary string `s = "001"`.

Firstly, we'll consider the two patterns we can alternate from:

- Pattern starting with '0': "010"
- Pattern starting with '1': "101"

Now, we will calculate mismatches for the pattern starting with '0':

- For index 0: The given character is '0' and matches the expected '0', so the mismatch count (`cnt`) remains 0.
- For index 1: The given character is '0' but the expected character is '1'. This is a mismatch and `cnt` becomes 1.
- For index 2: The given character is '1' and matches the expected '1', so `cnt` remains 1.

Having finished the loop, we know that `cnt` for the pattern "010" is 1. This means we would need a minimum of 1 operation to turn "001" into "010".

However, we need to check against the opposite pattern too, starting with '1':

- For index 0: The given character is '0' but we expect '1'. This is a mismatch, so we start with `cnt` 1.
- For index 1: The given character is '0' and since we're inverting our expectation, we want a '0' here. No mismatch, `cnt` remains 1.
- For index 2: The given character is '1' and matches our expected '0' (since we invert at even indices). This is a mismatch, so `cnt` becomes 2.

When we compare the number of operations needed, we have two possibilities:

- For the pattern starting with '0', `cnt` is 1. Therefore, we need 1 operation.
- For the pattern starting with '1', `cnt` is 2. But instead of flipping mismatches, we can flip the rest and achieve the same pattern, which means `len(s) - cnt = 3 - 2 = 1` operation too.

Both patterns would require the same number of operations, just 1 in this case. The answer to the example would be that we need a single operation to make the string alternating. Thus, applying the logic from the solution approach, we take the minimum of `cnt` and `len(s) - cnt`, which is 1.

Solution Implementation

Python

```python
class Solution:
    def minOperations(self, string: str) -> int:
        # Count the number of characters that do not match the pattern '01' repeated.
        # '01'[i & 1] generates '0' for even i and '1' for odd i, which is the expected pattern.
        mismatch_count = sum(char != '01'[index % 2] for index, char in enumerate(string))

        # Since there are only two possible patterns ('01' repeated or '10' repeated),
        # if mismatch_count is the cost of converting to pattern '01',
        # then len(string) - mismatch_count will be the cost of converting to pattern '10'.
        # We choose the minimum of these two costs as our result.
        return min(mismatch_count, len(string) - mismatch_count)
```

Java

```java
class Solution {
    public int minOperations(String s) {
        // 'count' variable counts how many operations are needed to make the string alternating (starting with '0')
        int count = 0;
        // The 'length' of the string
        int length = s.length();

        // Loop through each character in the string
        for (int i = 0; i < length; ++i) {
            // Check if the current character does not match the pattern '01' alternately
            // by checking the parity of the index i (even or odd) using bitwise AND with 1 (i & 1)
            if (s.charAt(i) != "01".charAt(i & 1)) {
                // If it does not match, increment the 'count' (operation needed)
                count += 1;
            }
            // Note: No need to do anything if it matches since no operation is needed
        }

        // The minimum operations would be the smaller of 'count' or the inverse operations needed (length - count)
        // This is to account for the possibility that starting with '1' might require fewer operations
        return Math.min(count, length - count);
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to return the minimum number of operations required
    // to make all the characters at odd indices equal to '0' and
    // even indices equal to '1' or vice versa in the given string 's'.
    int minOperations(string s) {
        int count = 0;              // Initialize count of operations.
        int length = s.size();      // Get the size of the string.

        // Iterate over the string characters.
        for (int i = 0; i < length; ++i) {
            // Check if the current character does not match the expected pattern.
            // The pattern is '01' for even indices and '10' for odd indices.
            // The expression "01"[i & 1] effectively toggles between '0' and '1'.
            count += s[i] != "01"[i & 1];
        }

        // Return the minimum of the count of changes if we start with '0'
        // and the count if we start with '1'. The latter is given by (length - count)
        // since it represents the number of positions that match the pattern when starting with '0'
        // and therefore do not match the pattern when starting with '1'.
        return std::min(count, length - count);
    }
};
```

TypeScript

```typescript
/**
 * Computes the minimum number of operations to make a binary string alternate
 * between '0' and '1' characters.
 * @param {string} binString - The input binary string to transform.
 * @return {number} The minimum number of operations required.
 */
function minOperations(binString: string): number {
    // n stores the length of the input binary string.
    const n: number = binString.length;
    // count will hold the number of changes needed to match '01' alternating pattern.
    let changeCount: number = 0;

    // Iterate over each character in the binary string.
    for (let i = 0; i < n; i++) {
        // If the current character does not match the '01' pattern, increment changeCount.
        // '01'[i & 1] will alternate between '0' for even i and '1' for odd i.
        // The expression (i & 1) is equivalent to (i % 2), but may be more efficient.
        if (binString[i] !== '01'[i & 1]) {
            changeCount++;
        }
    }

    // The result is the minimum between changeCount and the number of operations
    // to match the opposite pattern (starting with '1'). This is because we can
    // start with either '0' or '1', and we want the minimum of both options.
    return Math.min(changeCount, n - changeCount);
}
```

```python
class Solution:
    def minOperations(self, string: str) -> int:
        # Count the number of characters that do not match the pattern '01' repeated.
        # '01'[i & 1] generates '0' for even i and '1' for odd i, which is the expected pattern.
        mismatch_count = sum(char != '01'[index % 2] for index, char in enumerate(string))

        # Since there are only two possible patterns ('01' repeated or '10' repeated),
        # if mismatch_count is the cost of converting to pattern '01',
        # then len(string) - mismatch_count will be the cost of converting to pattern '10'.
        # We choose the minimum of these two costs as our result.
        return min(mismatch_count, len(string) - mismatch_count)
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input string `s`. This is because there is a single loop in the code iterating through the string, and within the loop, basic operations are performed which take constant time.

The space complexity of the code is `O(1)`. This is due to the use of a fixed amount of extra space, which is a few variables (`cnt` and computed values of `len(s) - cnt`), regardless of the input size.