2914. Minimum Number of Changes to Make Binary String Beautiful

Medium

Problem Description

You are given a binary string `s` that is 0-indexed and has an even length. The concept of a binary string is that it contains only two characters, '0' and '1'. A binary string is considered **beautiful** if it can be divided into one or more substrings where each substring satisfies two conditions. Firstly, it must have an even length, and secondly, it should only contain the character '1' or the character '0', not both. Your task is to make the string beautiful by possibly changing any of the characters in the string to either '0' or '1'. The goal is to achieve this with the least number of changes.

To assist in finding a solution, think about the characteristics of a beautiful substring. Since a beautiful substring can only have one type of character, it implies that we cannot have alternating characters. Therefore, we need to find a way to ensure that characters at specific positions are the same so that the substrings of even length have only '0's or only '1's.

Intuition

The solution is centered around dividing the string into even-length substrings that only have identical characters. To achieve the minimum number of changes, we try to identify where the string already conforms to the beautiful criteria and only change the characters that disrupt the pattern. Since every two consecutive characters must be the same, the strategy is to compare characters at all odd indices with their preceding even index neighbors.

We iterate through the string, checking characters at odd indexes, i.e., `1, 3, 5, ...`, comparing each to its previous even index character. If these two consecutive characters differ, `s[i] != s[i - 1]`, we recognize that we'd need to change the character at the odd index for the substring to be beautiful. For each necessary change, we increment the count.

Solution Approach

The implementation of the solution provided in the reference approach is straightforward and elegant due to its simplicity. It leverages a single pass counting method, which is an efficient algorithm for this problem. The choice of data structures or patterns used here is minimal as the solution primarily relies on accessing the characters of the string and performing comparisons.

The process begins by initializing a counter set to zero. This counter will keep track of the minimum number of changes required to make the binary string beautiful.

The core of the solution involves iterating through the string, but instead of looking at every character, we only examine characters at every odd index, which are `$1, 3, 5, \cdots$`. This approach works because beautiful substrings need to be in pairs (`00` or `11`), and any discrepancy in these pairs would be caught when comparing the odd-indexed character with its directly preceding even-indexed character.

During each iteration of the loop, we compare the character at the current index `i` to the one before it `s[i - 1]`. If the characters are different, `s[i] != s[i - 1]`, it signals that we would need to change the character at index `i` to make the pair of characters the same, thus contributing towards a beautiful substring.

We implement the condition described above using a generator expression summed up to accumulate the total:

```
sum(s[i] != s[i - 1] for i in range(1, len(s), 2))
```

Here's a breakdown of this expression:

- `range(1, len(s), 2)`: Creates a sequence of numbers starting from 1 up to the length of the string `s` in steps of 2. This sequence represents all odd indices up to the end of the string.
- `s[i] != s[i - 1]`: The logic to determine if a change is required. It evaluates to `True` (which is equivalent to `1`) when a change is needed, and `False` (which is equivalent to `0`) when it's not.
- `sum(...)`: Sums up the values generated by the expression for each `i`. Since `True` is equivalent to `1`, summing these values gives us the total number of changes required.

The result of the sum is the total count of changes needed, which we return as the solution to the problem. This approach ensures that we only change the minimum number of characters necessary for all pairs of characters in the string to become identical, thereby making the string beautiful with the least amount of effort.

Example Walkthrough

Let's consider a small binary string as an example to illustrate the solution approach: `s = "010101"`. We aim to find the minimum number of changes needed to make `s` beautiful.

With the solution approach in mind, we inspect the characters at the odd indices of `s` and compare each with the preceding even index character to identify where changes are needed.

1. `i = 1`: We compare `s[1]` with `s[0]`. Here, `s[1] = '1'` and `s[0] = '0'`. Since `s[1] != s[0]`, this indicates that we need to make one change to make the substring `s[0:2]` beautiful.
2. `i = 3`: We compare `s[3]` with `s[2]`. Here, `s[3] = '1'` and `s[2] = '0'`. Once again, we see that `s[3] != s[2]`, signaling the need for another change for the substring `s[2:4]` to be beautiful.
3. `i = 5`: Finally, we compare `s[5]` with `s[4]`. We find that `s[5] = '1'` and `s[4] = '0'`, which means `s[5] != s[4]`. This means a change is required for the substring `s[4:6]` to conform to the beautiful criteria.

In every comparison for this specific string, a change is needed. As such, the counter would increment by 1 for each of the three steps, resulting in a total of 3 changes necessary to transform `s` into a beautiful binary string. Specifically, we could change `s` to either "000000" or "111111". Either way, the minimum number of changes required is 3.

Using the solution approach's sum generator expression:

```
sum(s[i] != s[i - 1] for i in range(1, len(s), 2))
```

We calculate the result as:

- At `i = 1`, `s[1] != s[0]` yields `True` (`1`).
- At `i = 3`, `s[3] != s[2]` yields `True` (`1`).
- At `i = 5`, `s[5] != s[4]` yields `True` (`1`).

Therefore, the sum is `1 + 1 + 1 = 3`. This confirms that 3 is the minimum number of changes needed to make the string `s` beautiful, aligning with our step-by-step analysis.

Solution Implementation

Python

```python
class Solution:
    def minChanges(self, string: str) -> int:
        # Initialize a variable to keep track of the changes needed.
        changes_needed = 0

        # Iterate over the string starting from the second character until the end
        # and consider only odd indices (Python uses 0-based indexing).
        for i in range(1, len(string), 2):

            # Check if the current character is different from the previous one.
            # If so, this is not a change we are interested in, so continue.
            # We are interested in pairs of characters that are the same and
            # as given alternate characters should be different in a "nice" string.
            if string[i] == string[i - 1]:
                changes_needed += 1  # Increment the changes needed.

        # Return the total number of changes needed to have no two consecutive characters
        # being the same at odd indices, making the string "nice" as per the defined condition.
        return changes_needed
```

Java

```java
class Solution {
    // Method to find the minimum number of changes required to make all characters at even indices
    // the same as the previous characters at odd indices in a given string.
    public int minChanges(String s) {
        int changesNeeded = 0; // Initialize a counter for the number of changes needed

        // Iterate over the string starting from the second character, checking every other character
        for (int i = 1; i < s.length(); i += 2) {
            // If the current character is not the same as the previous character,
            // we need to change it to make it the same
            if (s.charAt(i) != s.charAt(i - 1)) {
                changesNeeded++; // Increment the counter for the number of changes needed
            }
        }

        // Return the total number of changes required to make the string's odd and even characters match
        return changesNeeded;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the minimum number of changes needed
    // to make the string alternate between two characters.
    int minChanges(string str) {
        int changesNeeded = 0; // Initialize the count of changes needed.
        int length = str.size(); // Get the size of the string.

        // Loop through the string, considering every second character starting from index 1,
        // to ensure alternation.
        for (int i = 1; i < length; i += 2) {
            // Check if the current character is the same as the previous one.
            // If they are the same, a change is needed to ensure alternation.
            // Increment the count of changes needed.
            changesNeeded += (str[i] != str[i - 1]);
        }

        // Return the total number of changes needed to make the string alternate.
        return changesNeeded;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum number of changes required to create a string without alternating characters.
 * It assumes that only every second character can be changed.
 *
 * @param {string} s - The string to be processed.
 * @return {number} The minimum number of changes required.
 */
function minChanges(s: string): number {
    // Initialize a variable to track the number of changes needed.
    let changesNeeded = 0;

    // Iterate through the string, checking every second character starting from the second one.
    for (let i = 1; i < s.length; i += 2) {
        // Check if the current character is different from the previous one.
        // If they are the same, increment the changes needed.
        if (s[i] !== s[i - 1]) {
            changesNeeded++;
        }
    }

    // Return the total number of changes needed.
    return changesNeeded;
}
```


```python
class Solution:
    def minChanges(self, string: str) -> int:
        # Initialize a variable to keep track of the changes needed.
        changes_needed = 0

        # Iterate over the string starting from the second character until the end
        # and consider only odd indices (Python uses 0-based indexing).
        for i in range(1, len(string), 2):

            # Check if the current character is different from the previous one.
            # If so, this is not a change we are interested in, so continue.
            # We are interested in pairs of characters that are the same and
            # as given alternate characters should be different in a "nice" string.
            if string[i] == string[i - 1]:
                changes_needed += 1  # Increment the changes needed.

        # Return the total number of changes needed to have no two consecutive characters
        # being the same at odd indices, making the string "nice" as per the defined condition.
        return changes_needed
```


Time and Space Complexity

The time complexity of the function `minChanges` is `O(n)`, where `n` is the length of the string `s`. This is because the function includes a loop that iterates over the length of the string, starting from the second character (index 1) to the end of the string, with increments of 2. In each iteration, the function calculates whether the current character is different from its predecessor, which is an `O(1)` operation. Therefore, the loop runs approximately `n/2` times, which still yields a linear time complexity of `O(n)`.