2712. Minimum Cost to Make All Characters Equal

Medium Greedy String Dynamic Programming

Leetcode Link

Problem Description

In this problem, we are dealing with a binary string `s`, which is a sequence of characters that can only be `0` or `1`. The string uses a 0-based index, meaning that the first character of the string is at position 0, the next at position 1, and so on, up to the last character, which is at position `n - 1`. Our goal is to transform this binary string into a uniform string where all characters are the same, either all `0`s or all `1`s.

To achieve a uniform string, we can perform two types of operations:

1. Select an index `i`, and flip all the characters from the beginning of the string (index 0) up to and including `i`. For instance, if we flip characters from index 0 to index `i` in `"01011"`, and `i` is 2, the string becomes `"10100"`. This operation comes with a cost equal to `i + 1`.

2. Select an index `i`, and flip all the characters from that index `i` to the end of the string (index `n - 1`). For example, for the same string `"01011"`, if we flip characters from index 2 to index 4 (`n - 1`), the string becomes `"01100"`. This operation has a cost of `n - i`.

Each flip inverts the characters, which means `0` becomes `1` and `1` becomes `0`. The challenge is to find the minimum total cost of such flipping operations that will result in a string composed entirely of the same character.

Intuition

The intuition behind the provided solution is to identify the positions in the string where flipping can actually contribute to making the string uniform and doing it at the lowest possible cost. If all characters are already equal, no operation is needed and the cost is zero. We only need to consider flipping at positions where a character differs from its preceding character, indicating a point of change in the string's structure.

For each character `s[i]` that differs from `s[i - 1]`, there is a possibility of flipping either the sub-string before `i` or the sub-string after `i` to make the characters equal. We want to choose the index that minimizes the cost for each situation, which is given by `min(i, n - i)` where `i` is the cost for flipping from the start and `n - i` is the cost for flipping until the end. By iterating through the string and summing up the minimum cost at each change point, we accumulate the total minimum cost needed to make the string uniform.

Here is the step-by-step intuition for the provided solution:

- Initialize `ans` as `0`, which will hold the overall minimum cost.
- Loop through each character in the string, starting from index 1 up to index `n - 1`.
- Check if the current character `s[i]` is different from the previous character `s[i - 1]`.
- When a difference is detected, it means we have a potential flip point.
- Calculate the cost to flip either from the start up to `i` or from `i` to the end, by evaluating `min(i, n - i)`.
- Add this cost to the `ans` total.
- After looping through the entire string, `ans` will contain the minimum cost to make the string uniform.
- Return the `ans` value.

The solution effectively avoids redundant operations and achieves minimal cost by flipping at the optimal points in the string.

Solution Approach

The implementation of the solution provided uses a simple linear scan algorithm which is both time and space efficient because it iterates over the string once and does not require any additional data structures. The key pattern used here is the iteration over changing points in the binary string.

Here's the breakdown of the solution approach:

1. **Initialization**: A variable `ans` is created to store the accumulated minimum cost and its initial value is set to `0`. The length of the string `n` is computed to avoid recalculating it during each iteration.

   ```
   ans, n = 0, len(s)
   ```

2. **Looping through the changes**: The program then iterates from `1` to `n - 1`, intentionally starting from `1` because we always compare the current character with the previous one to identify the change points.

   ```
   for i in range(1, n):
   ```

3. **Detecting change points**: The condition inside the loop checks if the current character `s[i]` is different from the preceding character `s[i - 1]`. If it is, then a flip operation must be performed at this point, because we want to eliminate the discrepancy to move towards a uniform string.

   ```
   if s[i] != s[i - 1]:
   ```

4. **Calculating operation costs**: For any change point, the cost of making the preceding or succeeding substring uniform is analyzed. Since the goal is to do this at the minimum cost, the function `min(i, n - i)` calculates the cheaper option between flipping the first `i` characters or the last `n - i` characters. This corresponds to choosing the shorter sub-string to flip which naturally costs less.

   ```
   ans += min(i, n - i)
   ```

   The result is added to our running total `ans`.

5. **Final result**: After all potential change points have been processed, the variable `ans` now contains the minimum cost to make all characters of the string equal. The function finally returns `ans`.

   ```
   return ans
   ```

By utilizing a single pass over the string and performing an efficient comparison and calculation at each step, the implementation ensures a minimal time complexity of O(n), where n is the length of the string. There's no use of additional data structures, so the space complexity is O(1). This approach ensures optimal performance for the problem at hand.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach. We have the following binary string `s = "0110101"`.

1. **Initialization**: Set the total minimum cost `ans` to `0` and compute the length of `s` to be `n = 7`.

   ```
   ans, n = 0, len(s)  # ans = 0, n = 7
   ```

2. **Looping through the changes**: Start iterating from index `1` to `n - 1`, which goes from index `1` to index `6`.

3. **Detecting change points**: We're looking for positions where `s[i] != s[i - 1]`. These are our points of change:

   - At `i = 1`, `s[i] = "1"` and `s[i - 1] = "0"`. This is a change point; `s[1]` differs from `s[0]`.
   - At `i = 2`, `s[i] = "1"` and `s[i - 1] = "1"`. This is not a change point; `s[2]` is the same as `s[1]`.
   - At `i = 3`, `s[i] = "0"` and `s[i - 1] = "1"`. This is another change point; `s[3]` differs from `s[2]`.
   - At `i = 4`, `s[i] = "1"` and `s[i - 1] = "0"`. Another change point; `s[4]` differs from `s[3]`.
   - At `i = 5`, `s[i] = "0"` and `s[i - 1] = "1"`. Yet another change point; `s[5]` differs from `s[4]`.
   - At `i = 6`, `s[i] = "1"` and `s[i - 1] = "0"`. The last change point; `s[6]` differs from `s[5]`.

4. **Calculating operation costs**: For every change point identified, calculate the cost:

   - At `i = 1`, the cost is `min(1, 7 - 1) = 1`. Add this to `ans` to get `ans = 1`.
   - At `i = 3`, the cost is `min(3, 7 - 3) = 3`. Now `ans = 1 + 3 = 4`.
   - At `i = 4`, the cost is `min(4, 7 - 4) = 3`. Update `ans = 4 + 3 = 7`.
   - At `i = 5`, the cost is `min(5, 7 - 5) = 2`. Update `ans = 7 + 2 = 9`.
   - At `i = 6`, the cost is `min(6, 7 - 6) = 1`. Finally, `ans = 9 + 1 = 10`.

5. **Final result**: After the loop, we have checked all change points. The total minimum cost `ans` is `10`. Thus, this is the minimum cost required to make the string `s = "0110101"` uniform, using the flip operations described.

By sequentially applying the logic to each change point, the example demonstrates how we sum up the minimum of flipping costs at every change point to get the total minimal cost efficiently.

Python Solution

```python
class Solution:
    def minimumCost(self, s: str) -> int:
        """
        Calculate the minimum cost of processing the string where
        cost is defined as the minimum distance from either end of the string
        to the position where the character differs from its adjacent one.

        :param s: The input string consisting of characters.
        :type s: str
        :return: The minimum cost of processing the string.
        :rtype: int
        """
        # Initialize the answer variable to store the total cost
        total_cost = 0

        # Get the length of the string
        n = len(s)

        # Iterate through the string starting from the second character
        for i in range(1, n):
            # Check if the current character is different from the previous character
            if s[i] != s[i - 1]:
                # If so, add the minimum distance from either end of the string
                total_cost += min(i, n - i)

        # Return the calculated total cost
        return total_cost
```

Java Solution

```java
class Solution {

    /**
     * Calculates the minimum cost to ensure no two adjacent characters are the same.
     *
     * @param s The input string.
     * @return The total minimum cost.
     */
    public long minimumCost(String s) {
        long totalCost = 0; // Holds the running total cost
        int lengthOfString = s.length(); // Stores the length of the string once for efficiency

        // Loop through each character in the string, starting from 1 as we're comparing it with the previous character
        for (int i = 1; i < lengthOfString; ++i) {
            // Check if the current character is different from the previous one
            if (s.charAt(i) != s.charAt(i - 1)) {
                // Calculate minimum cost for this char to be either from the start or end of the string
                totalCost += Math.min(i, lengthOfString - i);
            }
        }

        return totalCost; // Return the calculated total cost
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the minimum cost of operations to make the string stable
    long long minimumCost(string s) {
        long long cost = 0; // Initialize the total cost to 0
        int length = s.size(); // Get the length of the string

        // Iterate through the string starting from the second character
        for (int i = 1; i < length; ++i) {
            // Check if the current character is different from the previous one
            if (s[i] != s[i - 1]) {
                // If different, add the minimum of 'i' or 'length - i' to the cost
                // 'i' represents the cost to change all characters to the left to match the current one
                // 'length - i' represents the cost to change all characters to the right
                cost += min(i, length - i);
            }
        }
        // Return the calculated cost
        return cost;
    }
};
```

Typescript Solution

```typescript
// Function to calculate the minimum cost to make a binary string beautiful.
// A binary string is considered beautiful if it does not contain any substring "01" or "10".
function minimumCost(s: string): number {
    // Initialize the answer, which will store the minimum cost.
    let cost = 0;
    // Get the length of the string.
    const lengthOfString = s.length;

    // Iterate through the string characters starting from the second character.
    for (let index = 1; index < lengthOfString; ++index) {
        // If the current and previous characters are different,
        // It implies a "01" or "10" substring, which is not beautiful.
        if (s[index] !== s[index - 1]) {
            // The cost to remove this not-beautiful part is the minimum
            // of either taking elements from the left or the right of it.
            cost += Math.min(index, lengthOfString - index);
        }
    }

    // Return the calculated minimum cost to make the string beautiful.
    return cost;
}
```

Time and Space Complexity

Time Complexity

The given code consists of a single loop that iterates through the length of the input string `s`. During each iteration, it performs a constant number of operations (comparing characters and calculating the minimum of two numbers). Since the loop runs for `n-1` iterations (where `n` is the length of `s`), the time complexity is `O(n)`.

Space Complexity

The code uses a fixed number of integer variables (`ans`, `n`, and `i`). These variables do not depend on the size of the input string `s`, hence the space complexity is `O(1)` as no additional space is proportional to the input size.