2957. Remove Adjacent Almost-Equal Characters

Medium Greedy String Dynamic Programming

Problem Description

The given problem involves a string `word` where the objective is to make the minimum number of operations so that no pair of adjacent characters remain that are "almost-equal." A pair of characters are defined as "almost-equal" if either they are the same character or they are adjacent to each other in the alphabet (for example, 'b' and 'c' are adjacent alphabetically).

An operation consists of picking any character in `word` and changing it to any lowercase English letter of your choice. The task is to find out the least number of such operations required to ensure no "almost-equal" characters are next to each other in the string.

Intuition

To approach this solution efficiently, we utilize a greedy algorithm. The underlying thought process of a greedy approach is to solve a problem by making a sequence of choices, each of which simply looks the best at the moment. It doesn't reconsider choices previously made, even if future actions may create a suboptimal sequence.

Starting from the first character, we traverse the string and consider each adjacent pair of characters. If we find a pair that is almost-equal, we increment our operation counter because we'll need to change one of those characters to remove the almost-equality. We then skip checking the next character because it has been taken care of by this operation. If a pair is not almost-equal, we continue checking with the next character.

By doing this, we count only the minimal number of changes needed without actually performing the substitutions, which aligns precisely with what the problem is asking for—the least number of operations needed, not the resulting string.

Solution Approach

The implementation of the solution is straightforward and follows the greedy approach described in the intuition section. The code is written in Python and utilizes basic string manipulation and character comparison by their ASCII values.

Here's a breakdown of the algorithm used in the implementation:

1. Initialize a variable `ans` to `0` that will hold the count of necessary operations.
2. Start iterating over the string `word` from the second character (index `1` since the string is 0-indexed).
3. For each character at index `i`, compare it with the previous character at index `i - 1` by calculating the absolute difference of their ASCII values.
4. If the difference in ASCII value is less than `2`, it implies that the characters are either the same or adjacent in the alphabet, and thus "almost-equal".
5. In such a case, since we need to perform an operation to change either of the "almost-equal" characters, increment `ans` by `1`.
6. After performing an operation, we make a greedy choice to move `2` indices forward, because we know the next character does not need to be checked as it's already been part of an "almost-equal" pair and would be changed by the operation, thus `i` is incremented by `2`.
7. If the characters are not "almost-equal," simply move to the next character to continue the check by incrementing `i` by `1`.
8. Continue this process until all characters have been checked.
9. Finally, the value of `ans` provides the minimum number of operations needed to achieve the objective.

The solution doesn't make use of any complex data structures; it only operates on the given string and requires a single pass from left to right, checking the adjacent characters. This approach ensures that the time complexity of the solution is linear, i.e. `O(n)`, where `n` is the length of the given string. There are no additional memory requirements other than the variables for iteration and counting, making it a constant space solution, `O(1)`.

Example Walkthrough

Consider the string `word = "abaaccdbbd"`. We want to find the minimum number of operations needed so that no pair of adjacent characters are "almost-equal." Here is how the greedy solution approach is applied step by step:

1. Initialize `ans = 0`. No operations have been performed yet.
2. Start iterating over `word` from the second character.
3. Compare characters at index 0 and index 1, 'a' and 'b'. The ASCII difference is 1, they are adjacent alphabetically, so they are "almost-equal."
4. Increment `ans` to 1 and skip to comparing characters at index 2 and 3.
5. Compare characters at index 2 and 3, 'a' and 'a'. They are the same, which means "almost-equal."
6. Increment `ans` to 2 and skip to comparing characters at index 4 and 5.
7. Compare characters at index 4 and 5, 'c' and 'c'. They are the same.
8. Increment `ans` to 3 and skip to comparing characters at index 6 and 7.
9. Compare characters at index 6 and 7, 'd' and 'b'. The ASCII difference is not 1, they are not "almost-equal."
10. Move to the next characters and compare at index 7 and 8.
11. Compare characters at index 7 and 8, 'b' and 'b'. They are the same.
12. Increment `ans` to 4. Since we are at the penultimate character, the algorithm ends here.
13. The minimum number of operations needed is the final value of `ans`, which is 4.

The greedy approach only requires us to check each character once and move on based on the "almost-equal" condition, minimizing the number of operations effectively.

Solution Implementation

Python

```python
class Solution:
    def removeAlmostEqualCharacters(self, word: str) -> int:
        # Initialize a count for almost equal character pairs.
        count = 0

        # Initialize an index variable for iterating through the string.
        index = 1

        # Get the length of the word for boundary checks.
        length_of_word = len(word)

        # Loop through the word.
        while index < length_of_word:
            # Check if the absolute difference between the ASCII values of adjacent
            # characters is less than 2 (which means the characters are almost equal).
            if abs(ord(word[index]) - ord(word[index - 1])) < 2:
                # If so, increment the count since the pair is considered as removed.
                count += 1

                # Skip the next character as the pair has been "removed".
                index += 2
            else:
                # Move to the next character for comparison.
                index += 1

        # Return the total count of almost equal character pairs removed.
        return count
```

Java

```java
class Solution {
    // Method to remove "almost equal" characters from the given string 'word'
    public int removeAlmostEqualCharacters(String word) {
        int removalCount = 0; // This will store the count of the removals made
        int wordLength = word.length(); // Store the length of the word

        // Loop through the characters of the word while skipping one character after each removal
        for (int i = 1; i < wordLength; ++i) {
            // Check if the current character and the previous one are "almost equal"
            // "Almost equal" means their unicode difference is less than 2
            if (Math.abs(word.charAt(i) - word.charAt(i - 1)) < 2) {
                removalCount++; // Increment the count of near removals
                i++; // Skip the next character since we removed a pair
            }
        }

        // Return the total number of removals made
        return removalCount;
    }
}
```

C++

```cpp
class Solution {
public:
    int removeAlmostEqualCharacters(string word) {
        // 'count' variable is used to track the number of removals
        int count = 0;
        // Calculate the size of the string 'word'
        int wordSize = word.size();

        // Loop through the string starting from the second character
        for (int i = 1; i < wordSize; ++i) {
            // Check if the current and previous characters are almost equal (difference < 2)
            if (abs(word[i] - word[i - 1]) < 2) {
                // Increment count since these two characters will be removed
                ++count;
                // Skip the next character as it has been considered in the almost equal pair
                ++i;
            }
        }
        // Return the total number of removals
        return count;
    }
};
```

TypeScript

```typescript
// This function removes adjacent pairs of characters from a given word if their
// Unicode values differ by less than 2. It returns the number of character pairs removed.
function removeAlmostEqualCharacters(word: string): number {
    // Initialize the count of removed character pairs
    let removedCount = 0;

    // Loop through the string, starting from the second character
    for (let index = 1; index < word.length; ++index) {
        // Calculate the absolute difference between the current character and the previous one
        const charCodeDifference = Math.abs(word.charCodeAt(index) - word.charCodeAt(index - 1));

        // If the difference is less than 2, a pair is almost equal and should be removed
        if (charCodeDifference < 2) {
            // Increment the count of removed character pairs
            ++removedCount;
            // Skip the next character to only remove pairs
            ++index;
        }
    }

    // Return the count of removed character pairs
    return removedCount;
}


class Solution:
    def removeAlmostEqualCharacters(self, word: str) -> int:
        # Initialize a count for almost equal character pairs.
        count = 0

        # Initialize an index variable for iterating through the string.
        index = 1

        # Get the length of the word for boundary checks.
        length_of_word = len(word)

        # Loop through the word.
        while index < length_of_word:
            # Check if the absolute difference between the ASCII values of adjacent
            # characters is less than 2 (which means the characters are almost equal).
            if abs(ord(word[index]) - ord(word[index - 1])) < 2:
                # If so, increment the count since the pair is considered as removed.
                count += 1

                # Skip the next character as the pair has been "removed".
                index += 2
            else:
                # Move to the next character for comparison.
                index += 1

        # Return the total count of almost equal character pairs removed.
        return count
```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the string `word`. This is because the loop runs for at most `n` steps as it increments by either 1 or 2 in each iteration, ensuring that each character is visited no more than once.

The space complexity of the code is `O(1)`. It only uses a fixed number of extra variables (like `ans`, `i`, and `n`) that do not depend on the size of the input string, hence it uses constant additional space.