1544. Make The String Great

Easy Stack String

Problem Description

The problem requires us to cleanse a given string `s` consisting of both uppercase and lowercase English letters such that no two adjacent characters are a pair of the same letter in different cases (i.e., 'a' and 'A'). If such a pair exists, those two characters are considered bad and need to be removed. We start by identifying these bad pairs and continue removing them until there are no more such pairs, resulting in a 'good' string. The output is the 'good' string which does not contain any bad pairs. It's also noted that an empty string qualifies as a good string.

Intuition

The idea behind the given solution approach is to use a stack, which is a Last-In, First-Out data structure, to help detect and remove the bad pairs. Here's the step-by-step process to understand the intuition:

- Iterate through the given string character by character.
- If the current character, when paired with the last character in the stack (if any), does not form a bad pair, push it onto the stack.
- A 'bad pair' is defined as two adjacent characters where one is an uppercase letter and the other is a lowercase letter of the same kind (the difference in their ASCII values would be exactly 32 if one is uppercase and the other is lowercase).
- If a 'bad pair' is detected (checked by seeing if the ASCII value difference is 32), remove the top element from the stack, effectively deleting both characters from consideration.
- Continue this process until we have iterated through all characters in the string.
- Convert the stack to a string and return it as it represents the 'good' string free of bad pairs.

The key to this solution is recognizing that a stack is an ideal way to keep track of characters, as it allows us to easily add new characters or remove the last added character when we find a matching pair that should be eliminated.

Solution Approach

The solution employs a stack to manage the removal of bad pairs effectively. Here's a more detailed breakdown of the implementation:

1. Initialize an empty stack called `stk`.
2. Iterate through each character `c` in the input string `s`.
3. For each character:
   - Check if the stack is empty or not. Continue to the next step if it's not empty. If it is empty, push the current character onto the stack.
   - Calculate the absolute difference of ASCII values between the current character `c` and the character at the top of the stack `stk[-1]`.
   - If the absolute difference is 32, this means that the current character `c` and the top character on the stack are a bad pair. In this case, invoke `stk.pop()` to remove the last character from the stack, effectively eliminating the bad pair.
   - If the absolute difference is not 32, push the current character onto the stack, as it can't form a bad pair with the previous one.
4. Once the iteration is complete, all bad pairs would have been removed, and the stack will only contain the characters of the good string.
5. Convert the stack into a string by joining all characters in the stack using `"".join(stk)`.
6. Return the resulting good string.

Here is the code for the reference solution that implements this approach:

```
class Solution:
    def makeGood(self, s: str) -> str:
        stk = []  # A [stack](//problems/stack_intro) to store the characters
        for c in s:  # Iterate over each character in the string
            # If the stack is not empty and the current and top characters are a bad pair
            if stk and abs(ord(stk[-1]) - ord(c)) == 32:
                stk.pop()  # Remove the top character from the stack
            else:
                stk.append(c)  # Push the current character onto the stack
        return "".join(stk)  # Return the good string by joining the stack
```

This algorithm has a time complexity of O(n), where `n` is the length of the string, since we traverse the string once. The space complexity is O(n) as well in the worst case when there are no bad pairs, as all characters will be in the stack. However, in practice, the stack size will often be smaller than the input string because bad pairs will be removed during processing.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Given the input string `s = "aAbBcC"`, we aim to remove all bad pairs as per the rules.

Initialize an empty stack `stk`.

Iterate through the string:

1. For the first character `a`, the stack is empty, so we push `a` onto the `stk`.
2. For the second character `A`, we check with the top of the stack `stk[-1] = a`. The ASCII value difference between `a` and `A` is 32, so this is a bad pair. We pop `a` from the stack.
3. Now, we proceed to `b`. The stack is empty, so we push `b`.
4. Next, we go to `B`. It forms a bad pair with `b` (difference in ASCII value is 32). We pop `b` from the stack.
5. We move to `c`. The stack is empty, so `c` is pushed onto the stack.
6. Finally, for `C`, the stack contains `c`, and again, the ASCII values of `C` and `c` are 32 units apart. This is a bad pair, so we pop `c` from the stack.

At the end of the iteration, the stack is empty, indicating all characters formed bad pairs and have been removed.

The final 'good' string is then returned by joining all characters in the stack, which is an empty string in this case: `"".join(stk)`, hence the result is `""`.

This walk-through demonstrates how the stack helps in eliminating bad pairs efficiently, resulting in a cleaned string that contains no bad pairs, which is the expected output of the algorithm.

Solution Implementation

Python

```
class Solution:
    def makeGood(self, s: str) -> str:
        # Initialize a stack to keep track of characters
        stack = []

        # Iterate over each character in the provided string
        for char in s:
            # If the stack is not empty and the ASCII difference between
            # the current character and the last character in the stack is 32,
            # it means they are equal but with different cases (e.g., 'a' and 'A').
            if stack and abs(ord(stack[-1]) - ord(char)) == 32:
                # Since the characters are a bad pair (equal but different cases),
                # we pop the last character from the stack to "eliminate" the pair
                stack.pop()
            else:
                # If the stack is empty or the characters are not a bad pair,
                # push the current character onto the stack
                stack.append(char)

        # Join the characters in the stack to form the "good" string
        # and return it
        return "".join(stack)
```

Java

```
// Solution class to provide a method that makes the string "good" as per given conditions
class Solution {
    // Method to remove instances where a letter and its opposite case letter are adjacent
    public String makeGood(String str) {
        // StringBuilder to efficiently manipulate strings
        StringBuilder stringBuilder = new StringBuilder();

        // Loop through all characters in the input string
        for (char currentChar : str.toCharArray()) {
            // Check if stringBuilder is empty or if the last character does not cancel out with currentChar
            // The condition checks ASCII value difference: 32 is the difference between lower and uppercase letters
            if (stringBuilder.length() == 0 || Math.abs(stringBuilder.charAt(stringBuilder.length() - 1) - currentChar) != 32) {
                // If they do not cancel, append current character to stringBuilder
                stringBuilder.append(currentChar);
            } else {
                // If they cancel, remove the last character from stringBuilder
                stringBuilder.deleteCharAt(stringBuilder.length() - 1);
            }
        }
        // Return the "good" string after all cancellations are done
        return stringBuilder.toString();
    }
}
```

C++

```
class Solution {
public:
    // Function to remove characters that are the same letter but opposite case adjacent to each other.
    string makeGood(string s) {
        string stack; // Using a string to simulate a stack

        // Iterate over each character in the input string
        for (char c : s) {
            // If the stack is empty or the absolute difference
            // between the ASCII values of the top character in the stack
            // and the current character is not 32 (difference between lower case and upper case),
            // then push the current character onto the stack.
            if (stack.empty() || abs(stack.back() - c) != 32) {
                stack += c;
            } else {
                // If the top character of the stack is of opposite case but same letter as
                // the current character, pop the top character from the stack.
                stack.pop_back();
            }
        }

        // Return the resulting string after stack simulation, which is the "good" string.
        return stack;
    }
};
```

TypeScript

```
// Function to remove characters that are the same letter but opposite case adjacent to each other.
function makeGood(s: string): string {
    let stack: string = ''; // Using a string to simulate a stack

    // Iterate over each character in the input string
    for (let i = 0; i < s.length; i++) {
        const currentChar: string = s[i];

        // Check if the stack is not empty
        if (stack) {
            // Calculate the ASCII values difference between the top character of the stack and the current character
            const diff: number = Math.abs(stack.charCodeAt(stack.length - 1) - currentChar.charCodeAt(0));

            // If the absolute difference is 32 (difference between lower case and upper case),
            // then we found same letter but opposite case adjacent to each other.
            if (diff === 32) {
                // Remove the last character from the stack, which is equivalent to popping from the stack
                stack = stack.substring(0, stack.length - 1);
                continue; // Move to the next iteration
            }
        }

        // If the stack is empty or the letter cases do not cancel each other,
        // add the current character to the stack.
        stack += currentChar;
    }

    // Return the resulting string after stack simulation, which is the "good" string.
    return stack;
}

// You can now call the function with a string argument to use it.
const result: string = makeGood("aAbBcC");
console.log(result); // Expected output: ""
```

```
class Solution:
    def makeGood(self, s: str) -> str:
        # Initialize a stack to keep track of characters
        stack = []

        # Iterate over each character in the provided string
        for char in s:
            # If the stack is not empty and the ASCII difference between
            # the current character and the last character in the stack is 32,
            # it means they are equal but with different cases (e.g., 'a' and 'A').
            if stack and abs(ord(stack[-1]) - ord(char)) == 32:
                # Since the characters are a bad pair (equal but different cases),
                # we pop the last character from the stack to "eliminate" the pair
                stack.pop()
            else:
                # If the stack is empty or the characters are not a bad pair,
                # push the current character onto the stack
                stack.append(char)

        # Join the characters in the stack to form the "good" string
        # and return it
        return "".join(stack)
```

Time and Space Complexity

Time Complexity:

The time complexity of the given function is `O(n)`, where `n` is the length of the string `s`. Each character in the string is processed exactly once when it is iterated over and potentially processed a second time if it is popped from the stack. Since each character is pushed onto and popped from the stack at most once, the number of operations for each character is constant. Hence, the time complexity is linear with respect to the size of the input string.

Space Complexity:

The space complexity of the function is also `O(n)`. In the worst case, the stack `stk` might need to store all characters of the string if they are all unique and none of them is the opposite case of the neighboring character. For instance, an input like "abcdef" would result in all characters being pushed onto the stack before the function completes. Therefore, the space required is directly proportional to the input size, leading to a linear space complexity.