2696. Minimum String Length After Removing Substrings

Easy Stack String Simulation

Problem Description

In this problem, we are provided with a string `s` which contains only uppercase English letters. We are allowed to perform operations on this string, where one operation consists of removing any occurrence of the substrings `"AB"` or `"CD"` from `s`.

The goal is to determine the minimum possible length of the string `s` after applying any number of operations. An operation may create the opportunity to perform additional operations, as the removal of a substring could cause new occurrences of `"AB"` or `"CD"` to form by concatenation of the characters before and after the removed substring.

To note, it is not necessary that we remove all possible occurrences of `"AB"` or `"CD"` in one go. As we remove substrings, we need to keep checking if new occurrences appear, and remove those as well until no more removals are possible.

Intuition

The intuition behind the solution is to simulate the process of removing substrings iteratively in an efficient manner. A naïve approach might involve repeatedly scanning the string and removing occurrences of the target substrings which can be very inefficient.

A more efficient method is to use a stack data structure to keep track of the characters in the string as we iterate over it. We push each character onto the stack, but before we do that, we check the character on top of the stack. If the current character along with the character at the top of the stack forms a forbidden substring (`"AB"` or `"CD"`), we pop the top character from the stack instead of pushing the current character. This effectively removes the forbidden substring from the string.

By using a stack, we are processing each character only once, which makes this a linear time operation (O(n), where n is the length of the string `s`). The stack helps us to keep track of the characters that have potential to form a forbidden substring with future characters. At the end of the iteration, the characters left in the stack represent the final string after all possible removals of `"AB"` and `"CD"` substrings.

The length of the final string is the number of elements in the stack, minus one to account for the dummy element at the bottom of the stack that was used as an initial placeholder.

Solution Approach

To implement the solution, we use a simple yet powerful data structure: the stack. A stack is a collection of elements with two main operations — push (add an element to the top) and pop (remove the top element).

Here's a step-by-step explanation of how the code uses a stack to solve the problem:

1. We initialize an empty stack with a dummy character at the bottom: `stk = [""]`.
2. We then iterate over each character `c` in the input string `s`.
3. On encountering each character, we check the top element of the stack:
   - If the current character `c` is "B" and the last character in the stack is "A" (which means we found "AB"), we pop the last character from the stack.
   - If the current character `c` is "D" and the last character in the stack is "C" (which means we found "CD"), we pop the last character from the stack.
   - These conditions check for the presence of the forbidden substrings, `stk[-1]` refers to the top element of the stack.
4. If neither of the above conditions are met, it means the current character doesn't form a forbidden substring with the character at the top of the stack. So we push the current character onto the stack.
5. After the loop ends, the characters left in the stack represent the characters of the string after all the operations been applied. The stack does not include possible forbidden substrings formed by adjacent characters.
6. The length of the resultant string, which is also the minimum length possible after operations, is the size of the stack minus one (to account for the initial dummy character).
7. Finally, we return `len(stk) - 1`.

This implementation takes advantage of the stack pattern's Last-In-First-Out (LIFO) property to efficiently track and remove substrings that match the criteria as we traverse the string. The time complexity of this implementation is O(n), where n is the length of the string `s`, since we go through each character exactly once.

Example Walkthrough

Let's consider an example to illustrate the solution approach with the input string `s = "ACBDACBD"`.

Here's how the algorithm progresses:

1. **Initialize the stack**: Start with an empty stack with a dummy character. `stk = [""]`.
2. **Process first character "A"**: Since "A" doesn't form "AB" or "CD" with the dummy character, we push "A" onto the stack. `stk = ["", "A"]`.
3. **Process second character "C"**: "C" also doesn't complete any forbidden substring, so we push it. `stk = ["", "A", "C"]`.
4. **Process third character "B"**: Now, "B" with the preceding "C" does not form a forbidden substring. Therefore, push "B". `stk = ["", "A", "C", "B"]`.
5. **Process fourth character "D"**: "D" after "B" does not form a forbidden substring either, so we push "D". `stk = ["", "A", "C", "B", "D"]`.
6. **Process fifth character "A"**: No forbidden substring with "D", so "A" is pushed. `stk = ["", "A", "C", "B", "D", "A"]`.
7. **Process sixth character "C"**: As "C" follows "A", nothing is removed, and we push "C". `stk = ["", "A", "C", "B", "D", "A", "C"]`.
8. **Process seventh character "B"**: "B" is coming after "C", which does form the forbidden substring "CB" (Note: Only "AB" and "CD" are forbidden, not "CB".). Push "B". `stk = ["", "A", "C", "B", "D", "A", "C", "B"]`.
9. **Process eighth character "D"**: Finally, we encounter "D" after "B". The "CD" forms a forbidden substring, so rather than pushing "D", we pop "C" from the stack. Now, the stack is `stk = ["", "A", "C", "B", "D", "A", "B"]`.

After iterating through the entire string, we are left with the stack containing the elements from our original string minus any "AB" or "CD" substrings. The length of our final reduced string is `len(stk) - 1 = 6`.

Therefore, the minimum possible length of the string `s` after applying the operations is `6`.

Solution Implementation

Python

```python
class Solution:
    def minLength(self, s: str) -> int:
        # Initialize a stack with an empty string to simplify edge cases
        # such as when the input string is empty or when we need to check
        # the top element without additional null checks.
        stack = [""]

        # Iterate through each character 'c' in the input string 's'
        for c in s:
            # If the condition is met for removing characters, i.e.,
            # - if current character is 'B' and the top of the stack is 'A', or
            # - if current character is 'D' and the top of the stack is 'C',
            # then pop the top element from the stack as the pair is valid
            # for removal, according to the problem's conditions.
            if (c == "B" and stack[-1] == "A") or (c == "D" and stack[-1] == "C"):
                stack.pop()
            else:
                # If the current character does not meet the conditions
                # for removal, then add ('push') this character onto the stack.
                stack.append(c)

        # After processing all characters, the stack will contain the characters
        # that cannot be removed by the rules of the problem. Since we started
        # with an empty string in the stack, we subtract 1 to get the actual
        # number of characters in the final string.
        return len(stack) - 1

# The class method 'minLength' can be used by creating an instance of the Solution class
# and calling 'minLength' with a specific string. For example:
# solution = Solution()
# result = solution.minLength("ACBDAC")
# print(result) will print the length of the string after removal operations.
```

Java

```java
class Solution {
    // Method to determine the minimum length of the string
    // after performing the given reduction operations.
    public int minLength(String s) {
        // Initialize a stack to hold characters.
        Deque<Character> stack = new ArrayDeque<>();
        // Add a space as a sentinel to the stack to avoid empty stack checks.
        stack.push(' ');

        // Iterate over each character in the string.
        for (char currentChar : s.toCharArray()) {
            // Check for the conditions to pop the top of the stack.
            // If the current character is 'B' and the top is 'A', or
            // if the current character is 'D' and the top is 'C', then pop.
            if ((currentChar == 'B' && stack.peek() == 'A') || (currentChar == 'D' && stack.peek() == 'C')) {
                stack.pop();
            } else {
                // Push the current character onto the stack if no pair is found.
                stack.push(currentChar);
            }
        }
        // Return the stack size minus one to exclude the initial sentinel space.
        return stack.size() - 1;
    }
}
```

C++

```cpp
class Solution {
public:
    // Method to calculate the minimum length of the string after performing reductions.
    int minLength(string s) {
        // Initialize a stack represented by a string with an extra space to handle empty stack case.
        string stack = " ";

        // Iterate over each character in the input string.
        for (char& c : s) {
            // Check for reduction conditions: 'B' with 'A', or 'D' with 'C'.
            // If the condition is satisfied, pop the last character from the 'stack'.
            if ((c == 'B' && stack.back() == 'A') || (c == 'D' && stack.back() == 'C')) {
                stack.pop_back();
            } else {
                // If no reduction is possible, push the current character onto the 'stack'.
                stack.push_back(c);
            }
        }

        // Return the size of the stack minus one for the extra space at the beginning.
        // This gives the minimum length of the string after performing reductions.
        return stack.size() - 1;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum length of a string after removing specific pairs.
 * Pairs 'AB' and 'CD' are removed when encountered in the string.
 * @param {string} inputString - The original string.
 * @return {number} - The minimum length of the string after removals.
 */
function minLength(inputString: string): number {
    // Initialize a stack with an empty string to simplify checks
    const stack: string[] = [''];

    // Loop through each character in the input string
    for (const character of inputString) {
        // Peek at the last element of the stack
        const lastElement = stack[stack.length - 1];

        // Check for and handle 'AB' pair
        if (character === 'B' && lastElement === 'A') {
            stack.pop();
        }
        // Check for and handle 'CD' pair
        else if (character === 'D' && lastElement === 'C') {
            stack.pop();
        }
        // If no pairs are found, push the current character to the stack
        else {
            stack.push(character);
        }
    }

    // Subtract one since we added an empty string at the start
    return stack.length - 1;
}
```


```python
class Solution:
    def minLength(self, s: str) -> int:
        # Initialize a stack with an empty string to simplify edge cases
        # such as when the input string is empty or when we need to check
        # the top element without additional null checks.
        stack = [""]

        # Iterate through each character 'c' in the input string 's'
        for c in s:
            # If the condition is met for removing characters, i.e.,
            # - if current character is 'B' and the top of the stack is 'A', or
            # - if current character is 'D' and the top of the stack is 'C',
            # then pop the top element from the stack as the pair is valid
            # for removal, according to the problem's conditions.
            if (c == "B" and stack[-1] == "A") or (c == "D" and stack[-1] == "C"):
                stack.pop()
            else:
                # If the current character does not meet the conditions
                # for removal, then add ('push') this character onto the stack.
                stack.append(c)

        # After processing all characters, the stack will contain the characters
        # that cannot be removed by the rules of the problem. Since we started
        # with an empty string in the stack, we subtract 1 to get the actual
        # number of characters in the final string.
        return len(stack) - 1

# The class method 'minLength' can be used by creating an instance of the Solution class
# and calling 'minLength' with a specific string. For example:
# solution = Solution()
# result = solution.minLength("ACBDAC")
# print(result) will print the length of the string after removal operations.
```


Time and Space Complexity

The given code snippet defines a function `minLength` which processes a string `s` by simulating a stack to remove certain pairs of adjacent characters ('B' with a preceding 'A', and 'D' with a preceding 'C'). The algorithm uses a stack `stk` with a sentinel value (an empty string) pre-loaded, thus avoiding empty stack checks during the iteration.

Time Complexity:

The time complexity of this function is `O(n)` where `n` is the length of the input string `s`. This is because the function goes through each character in the string exactly once. At each character, it performs a constant-time check (`c == "B"` with `stk[-1] == "A"` or `c == "D"` with `stk[-1] == "C"`) and either pushes the character to the stack or pops from the stack. Both operations, pushing to and popping from the end of a list (which is used to simulate the stack in Python), occur in `O(1)` time. Thus, the overall time taken is proportional to the length of the string.

Space Complexity:

The space complexity of the algorithm can also be considered as `O(n)`. This is because in the worst-case scenario, none of the characters can be popped off the stack, and the stack would grow to contain all `n` characters from the input string. This would happen when there are no qualifying pairs ('B' with 'A' or 'D' with 'C') in the entire string. The sentinel value in the stack does not affect the complexity as it is a constant space overhead.

In summary:

- Time Complexity: `O(n)`
- Space Complexity: `O(n)`