921. Minimum Add to Make Parentheses Valid

Medium Stack Greedy String Leetcode Link

Problem Description

The problem presents a situation where we have a string `s` which consists only of the parentheses characters "(" and ")". A valid parentheses string is defined as such if it meets one of the following criteria:

- It's an empty string.
- It can be separated into two substrings `A` and `B` such that both `A` and `B` are themselves valid parentheses strings.
- It can be enclosed within a pair of parentheses — meaning if `A` is a valid string, then `(A)` is also valid.

The task at hand is to determine the fewest number of moves required to make the initial string `s` into a valid parentheses string. Each move consists of inserting exactly one parenthesis character (either "(" or ")") at any position within the string.

Intuition

The intuition behind the solution stems from the understanding of how a valid parentheses string is structured. Fundamentally, for every opening parenthesis "(", there should be a corresponding closing parenthesis ")" to make it valid. If we traverse the string from left to right, at any point, the number of closing parentheses should not exceed the number of opening parentheses.

When encountering an opening parenthesis, it suggests the start of a potentially valid substring, so we increment a counter. Upon finding a closing parenthesis, if there is a previously unmatched opening parenthesis (our counter is greater than zero), we can pair this closing parenthesis with that opening parenthesis, decrementing the counter. If the counter is zero (indicating no unmatched opening parentheses), we require an additional opening parenthesis to match the current closing parenthesis, thus incrementing the answer—keeping track of moves needed.

Finally, after running through the string, if there's any unmatched opening parenthesis remaining (counter is not zero), those need matching closing parentheses. So, we add the number of remaining unmatched opening parentheses to the answer.

By accumulating the moves required to insert missing opening or closing parentheses, we can calculate the minimum number of moves to make the string valid.

Solution Approach

The algorithm implemented in the solution is relatively straightforward and efficient, using a single pass through the string, and it relies on the use of counters to track the state of the parentheses. No additional data structures are needed, which makes the space complexity constant, `O(1)`, as we only use a couple of integer variables to keep track of counts. The algorithm can be described as follows:

1. Initialize two variables, `ans` and `cnt` to zero. Here, `ans` will hold the total number of moves required to make the string valid, and `cnt` will keep track of the balance of opening and closing parentheses as we iterate through the string.

2. Iterate through each character `c` of the given string `s`:
    - If `c` is an opening parenthesis `"("`, increment the `cnt` counter, since we have an additional unmatched opening parenthesis.
    - If `c` is a closing parenthesis `")"`:
        - If `cnt` is greater than zero, it means there is a preceding unmatched opening parenthesis which can be paired with this closing parenthesis, so we decrement `cnt`.
        - If `cnt` is zero, it means there are no unmatched opening parentheses to pair with this closing parenthesis, therefore, we need to add an opening parenthesis, thus we increment the `ans` counter.
3. After the iteration is complete, if there is a non-zero `cnt`, this means there are `cnt` number of unmatched opening parentheses remaining. These will all need matching closing parentheses, so we add `cnt` to `ans`.

4. The final value of `ans` is the minimum number of moves required to make the string valid.

This approach works well as it leverages the inherent structure of valid parentheses strings. Since we only look at the balance between opening and closing brackets without necessarily considering their exact positions, the order in which we would insert the additional parentheses doesn't change the number of moves we need to make, making this approach both simple and effective. The time complexity of the solution is linear, `O(n)`, where `n` is the length of the input string because we go through the string exactly once.

Example Walkthrough

Let's consider a small example here with the string `s = ")))((("` to illustrate the solution approach.

1. We initialize two variables, `ans` = 0 and `cnt` = 0.
2. We start iterating through the string `s` from left to right.
    - We begin with the first character `)`:
        - `cnt` is at 0, meaning there are no unmatched opening `(` for this `)`. We would need one opening `(` before this one to balance it out, so we increment `ans` to 1. Now, `ans` = 1 and `cnt` remains 0.
    - The second character is `)`: the same logic applies, so `ans` is incremented again. Now, `ans` = 2, and `cnt` is still 0.
    - For the third character `)`, we repeat the same increment on `ans`. So, `ans` = 3, and `cnt` = 0.
    - We now encounter an opening parenthesis `(`. We increment `cnt` by 1 because we have one unmatched opening parenthesis, so `cnt` = 1.
    - The fifth character is another `(`, so now `cnt` = 2.
    - The sixth character is yet again `(`, bringing `cnt` to 3.
3. We have finished iterating through the string. Now we must account for the unmatched opening parentheses. We have `cnt` = 3 unmatched `(` left, which means we need 3 matching closing `)` to make the string valid.
4. We add `cnt` to `ans`. So, `ans` becomes 3 + 3 = 6.

The minimum number of moves required to make the string `)))(((` valid by insertions is `ans` = 6. We can achieve this by inserting three opening `(` at the beginning and three closing `)` at the end, resulting in a valid parentheses string `((()))`.

Python Solution

```python
class Solution:
    def minAddToMakeValid(self, s: str) -> int:
        # Initialize variables to count the necessary additions and
        # keep track of the unmatched parentheses.
        additions_needed = unmatched_open = 0

        # Iterate through each character in the string.
        for char in s:
            # If the character is an open parenthesis, increment the unmatched count.
            if char == '(':
                unmatched_open += 1
            # If it's a close parenthesis and there's an unmatched open, pair it and decrement.
            elif unmatched_open:
                unmatched_open -= 1
            # Otherwise, if there is no unmatched open, we need an addition (an open parenthesis).
            else:
                additions_needed += 1

        # Add any remaining unmatched open parentheses to the total additions needed.
        additions_needed += unmatched_open

        # Return the total number of additions needed to make the string valid.
        return additions_needed
```

Java Solution

```java
class Solution {
    public int minAddToMakeValid(String s) {
        int additionsRequired = 0; // Count of parentheses to add to make the string valid
        int balanceCount = 0;      // Keep track of the balance between opening and closing brackets

        // Loop through each character in the string
        for (char character : s.toCharArray()) {
            if (character == '(') {
                // An opening parenthesis increments the balance count
                balanceCount++;
            } else if (balanceCount > 0) {
                // A closing parenthesis decrements the balance count if there are unmatched opening ones
                balanceCount--;
            } else {
                // If there are no unmatched opening, we need an opening parenthesis
                additionsRequired++;
            }
        }

        // Add the remaining unmatched opening parentheses to the count of required additions
        additionsRequired += balanceCount;

        // Return the total number of additions required to make the string valid
        return additionsRequired;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minAddToMakeValid(string s) {
        int balance = 0; // This will keep track of the balance between '(' and ')'
        int additions = 0; // Counter for the required additions to make the string valid

        // Loop through each character in the string
        for (char c : s) {
            // If it's an opening bracket, increase the balance
            if (c == '(') {
                balance++;
            }
            // If it's a closing bracket, check if there is a matching opening bracket
            else {
                // If there is a matching opening bracket, decrement the balance
                if (balance > 0) {
                    balance--;
                }
                // If there is no matching opening bracket, increment additions
                else {
                    additions++;
                }
            }
        }

        // Add outstanding balance to the additions. These are unmatched opening brackets.
        additions += balance;

        // Return the total number of additions required to make the string valid
        return additions;
    }
};
```

Typescript Solution

```typescript
// Function to calculate the minimum number of additions needed to make the parentheses string valid
function minAddToMakeValid(s: string): number {
    let balance = 0; // This will keep track of the balance between '(' and ')'
    let additions = 0; // Counter for the additions required to make the string valid

    // Loop through each character in the string
    for (let i = 0; i < s.length; i++) {
        const c = s[i];

        // If it's an opening bracket, increase the balance
        if (c === '(') {
            balance++;
        } else { // Implicitly c is ')', as it's not '('
            // If there is a matching opening bracket, decrement the balance
            if (balance > 0) {
                balance--;
            } else {
                // If there is no matching opening bracket, increment additions
                additions++;
            }
        }
    }

    // Add any unmatched opening brackets to the additions
    additions += balance;

    // Return the total number of additions required to make the string valid
    return additions;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the input string `s`. This is because the code iterates through each character of the string exactly once, executing a constant number of operations for each character.

Space Complexity

The space complexity of the provided code is `O(1)`, which signifies constant space. This is due to the fact that the space required does not scale with the size of the input string. The variables `ans` and `cnt` require a fixed amount of space regardless of the length of `s`.