856. Score of Parentheses

Medium Stack String

Problem Description

The problem presents us with a string `s` that contains only balanced parentheses. A balanced parentheses string means every opening parenthesis '(' has a corresponding closing parenthesis ')'. Our goal is to compute the "score" of this string based on certain rules:

1. A pair of parentheses "()" directly next to each other has a score of 1.
2. The score `AB` for two consecutive balanced parentheses strings `A` and `B` is the sum of their individual scores, i.e., `score(A) + score(B)`.
3. The score `(A)` for a balanced parentheses string `A` enclosed within another pair of parentheses is double the score of `A`, i.e., `2 * score(A)`.

The problem asks us to compute the total score of the given string based on these rules.

Intuition

To solve this problem, we can iterate through the string while keeping track of two things: the depth of nested parentheses `d`, and the current score `ans`. The depth increments each time we encounter an opening parenthesis '(' and decrements with a closing parenthesis ')'. The key insight is that a single pair of parentheses "()" contributes score based on the depth of nesting at that point; specifically, it contributes a score of 2^d, where `d` is the depth just before the pair ends.

Here is the thought process to arrive at the solution:

- Initialize the answer variable `ans` to zero and depth variable `d` to zero.
- Loop through each character `c` in the string `s`, using its index `i` to access previous characters if needed.
- If we encounter an opening parenthesis '(', we increase the depth `d` by 1, indicating we are going deeper in the nested structure.
- If we encounter a closing parenthesis ')', this means we are completing a parenthetical group; so we decrease the depth `d` by 1.
- Now, if the closing parenthesis is immediately following an opening one, i.e., we have a pair "()", it's time to add the score.
  - The tricky part is that if this pair is nested inside other pairs, its contribution to the score is not merely 1 but is actually 2^d where `d` is the depth just before this pair.
  - This is because each layer of nesting effectively doubles the score of the pair inside it.
  - We efficiently calculate 2^d as `1 << d` because bit shifting to the left is equivalent to multiplying by two.
  - We then add this computed score to `ans`.
- After processing all characters, `ans` holds the final score of the string, which we return.

This way, by using a depth variable and a bit of bitwise arithmetic, we compute the score in a single pass over the input string.

Solution Approach

The provided solution takes advantage of a stack-like behavior but optimizes it by using a depth variable instead to keep track of the levels of nested parentheses. This means that rather than actually pushing and popping from a stack, we can simply increment and decrement a depth counter to simulate the stack's depth tracking behavior.

Here's a step-by-step walkthrough of the implementation using the Reference Solution Approach:

1. **Initialization**: We initialize two integer variables:
   - `ans`: To store the cumulative score of the balanced parentheses string.
   - `d`: To keep track of the depth of the nested parentheses.
2. **Traversal**: We iterate through each character `c` of the string `s`. The loop utilizes `enumerate` to gain both the index `i` and the character `c`.
3. **Depth Update**:
   - When we encounter an opening parenthesis '(', we are going deeper into a level of nesting, so we increase `d` by 1.
   - Conversely, for a closing parenthesis ')', we decrease `d` by 1 as we are coming out of a nested level.
4. **Score Calculation**:
   - The crucial observation for score computation is made when we encounter a closing parenthesis ')'.
   - We check if the previous character `(s[i - 1])` was an opening parenthesis '('. If it was, then we have found a "()" pair.
   - The score of this pair is 2^d (which is `2` raised to the power of `d`). This is because each additional depth level doubles the score of a pair of parentheses.
   - We add this score to our answer `ans`.
5. **Result**: After the loop completes, `ans` holds the total score of the string which is then returned.

Algorithm: The algorithm is effectively a linear scan with a depth counting approach. The algorithm runs in O(n) time, with n being length of the string `s`, as each character is visited once.

Data Structures: While a stack is the intuitive choice for dealing with nested structures, this solution optimizes space complexity by using a single integer for depth tracking instead.

Patterns: This is an example of a single-pass algorithm with a small space optimization. The bit shifting trick (`1 << d`) is a clever way to compute powers of two that leverages the binary representation of integers.

This approach is efficient both in terms of time and space complexity, performing the calculation in linear time while using constant space.

Example Walkthrough

Let's illustrate the solution approach using a small example string: `"(()())"`.

1. **Initialization**:
   - `ans = 0`: To store the cumulative score.
   - `d = 0`: To keep track of the depth of nested parentheses.
2. **Traversal**:
   - We start with the first character '('.
     - `d` becomes `1` because we have entered a new level of nesting.
   - Next character '(':
     - `d` becomes `2` as we go deeper.
   - We encounter a closing parenthesis ')':
     - `d` is `2`, hence before decreasing `d`, we check the previous character which is '('.
     - Since it forms a pair "()", we add `1 << (d - 1)` to `ans` which is `1 << 1` = `2`.
     - `ans = ans + 2` and `d` decreases to `1`.
   - The next character is ')' again:
     - `d` is `1`, and we check the previous character which is not '(', so we just decrease `d` by `1`.
     - `ans` stays the same since we don't have a new pair, and `d` decreases to `0`.
   - Next character '(':
     - `d` becomes `1` since we are starting a new nesting level.
   - Lastly, we encounter ')':
     - `d` is `1`, the previous character is '(', so this is a pair "()", and we add `1 << (d - 1)` which is `1 << 0` = `1` to `ans`.
     - `ans = ans + 1` and `ans` becomes `3`, then `d` goes back to `0`.
3. **Traversal Complete**:
   - We have processed all characters, and our final score `ans` is `3`.

This example verifies our approach to scoring the string. Each pair of parentheses that forms "()" directly contributes `1 << (depth - 1)` to the score, with the depth being counted just before the closing parenthesis of the pair.

Solution Implementation

Python

```python
class Solution:
    def scoreOfParentheses(self, s: str) -> int:
        # Initialize the score and the current depth of the parentheses.
        score = current_depth = 0

        # Enumerate over the string to process each character along with its index.
        for index, char in enumerate(s):
            if char == '(':  # If the character is an opening parenthesis,
                current_depth += 1  # increase the depth.
            else:  # If it's a closing parenthesis,
                current_depth -= 1  # decrease the depth.
                # If the previous character was an opening parenthesis,
                # it's a pair "()". Add 2^depth to the score.
                if s[index - 1] == '(':
                    score += 1 << current_depth

        # Return the final computed score.
        return score
```

Java

```java
class Solution {
    public int scoreOfParentheses(String s) {
        // Initialize score and depth variables
        int score = 0;
        int depth = 0;

        // Iterate over the string
        for (int i = 0; i < s.length(); ++i) {
            // Check if the current character is an opening parenthesis
            if (s.charAt(i) == '(') {
                // Increase depth for an opening parenthesis
                ++depth;
            } else {
                // Decrease depth for a closing parenthesis
                --depth;
                // If the current character and the previous one are "()",
                // it's a balanced pair which should be scored based on the current depth
                if (s.charAt(i - 1) == '(') {
                    // Score the pair and add it to the total score
                    // 1 << depth is equivalent to 2^depth
                    score += 1 << depth;
                }
            }
        }
        // Return the final computed score
        return score;
    }
}
```

C++

```cpp
class Solution {
public:
    int scoreOfParentheses(string S) {
        // Initialize the score 'score' and the depth of the parentheses 'depth'
        int score = 0;
        int depth = 0;

        // Loop through each character in the string
        for (int i = 0; i < S.size(); ++i) {
            if (S[i] == '(') {
                // Increase depth for an opening parenthesis
                ++depth;
            } else {
                // Decrease depth for a closing parenthesis
                --depth;
                // Check if the previous character was an opening parenthesis
                // which reoresents the score of a balanced pair "()"
                if (S[i - 1] == '(') {
                    // Add to score: 2^depth, where depth is the
                    // depth of nested pairs inside the current pair
                    score += 1 << depth;
                }
            }
        }
        // Return the final computed score
        return score;
    }
};
```

TypeScript

```typescript
let score: number = 0; // Global score variable
let depth: number = 0; // Global depth variable to track the depth of parentheses

function scoreOfParentheses(S: string): number {
    score = 0; // Initialize score
    depth = 0; // Initialize depth

    // Loop through each character of the string
    for (let i = 0; i < S.length; i++) {
        if (S[i] === '(') {
            // If character is '(', increase the depth as this signifies entering a deeper level of nested parentheses
            depth++;
        } else {
            // If character is ')', decrease the depth as this signifies exiting from the current level
            depth--;

            // Check if the previous character was '(' to find a '()' pair which contributes to the score
            if (S[i - 1] === '(') {
                // The score of a pair is 2 to the power of its depth in the overall structure
                score += 1 << depth;
            }
        }
    }
    // Return the computed score after processing the entire string
    return score;
}

// The function scoreOfParentheses can now be used without a class, like so:
// let result = scoreOfParentheses("(()(()))");
```

```python
class Solution:
    def scoreOfParentheses(self, s: str) -> int:
        # Initialize the score and the current depth of the parentheses.
        score = current_depth = 0

        # Enumerate over the string to process each character along with its index.
        for index, char in enumerate(s):
            if char == '(':  # If the character is an opening parenthesis,
                current_depth += 1  # increase the depth.
            else:  # If it's a closing parenthesis,
                current_depth -= 1  # decrease the depth.
                # If the previous character was an opening parenthesis,
                # it's a pair "()". Add 2^depth to the score.
                if s[index - 1] == '(':
                    score += 1 << current_depth

        # Return the final computed score.
        return score
```

Time and Space Complexity

Time Complexity

The time complexity of the function `scoreOfParentheses` is `O(n)`, where `n` is the length of the string `s`. This is because there is a single loop that iterates through each character of the string `s` exactly once.

Space Complexity

The space complexity of the function is `O(1)`. The only extra space used is for the variables `ans` and `d`, which store the cumulative score and the current depth respectively. The space used does not scale with the size of the input string, so the space complexity is constant.