1541. Minimum Insertions to Balance a Parentheses String

Medium Stack Greedy String

Leetcode Link

Problem Description

The given problem involves a string `s` which contains only parentheses – specifically, the characters '(' and ')'. The objective is to make the string balanced according to the following rules:

- A left parenthesis '(' must be followed by two consecutive right parentheses '))'.
- The order must be maintained, i.e., for every '(', the corresponding ')' must come after it.

We can add parentheses anywhere in the string to balance it. The task is to find the minimum number of insertions required to balance the string.

Examples of balanced strings as per these rules are: "())", "(())))", and "(()())))". On the other hand, strings like ")(", "())", and "(()))" are not balanced as per the rules described.

Intuition

The intuition behind the solution is to iterate through the string, keeping track of how many opening and closing parentheses are needed to balance the string as we go.

We maintain two counters:

- `ic` to count the number of opening parentheses '(' we have seen that need right parentheses '))'.
- `ans` to count the number of insertions needed to balance the string.

While iterating through the string:

- When we encounter an '(', we increment `x` because we anticipate needing two more ')' to balance it later.
- When we encounter a ')', we have two cases:
  - If it's followed by another ')', this means we found a pair '))', so we decrement `x` as we have matched one opening '(' with two closing ')'.
  - If it's not followed by another ')', we save a single ')', so we add one to `ans` as we need to insert an additional ')' to balance the string, and then decrement `x`.
- If `x` is zero and a closing ')' without a matching '(' is found, we need to insert an opening '(' before it, so we add one to `ans`.
- After processing the closing parentheses, if `x` is greater than zero, it means there are unmatched '(' pending, so we need to add two ')' for each of them. Thus, we increment `ans` by `x` shifted to the left by 1 (which is equivalent to multiplying `x` by 2).

This solution ensures that we make the minimum number of insertions required to balance the string by only adding parentheses when necessary to make the string adhere to the balancing rules.

Solution Approach

The implementation of the solution involves a greedy approach to satisfy the conditions for a balanced parentheses string, as mentioned earlier. The algorithm can be detailed as follows:

1. **Initialize Two Counters:** We have two counters `ans` and `x`, where `ans` keeps track of the number of insertions needed and `x` keeps track of the number of unmatched opening parentheses that are yet to be paired with a closing parentheses.
2. **Iterating Over the String:** We iterate over each character in the string using a `while` loop, indexed by `i`.
3. **Handling Opening Parentheses '(':** When we encounter an opening parenthesis, we increment `x` by 1, as we need to find or insert two consecutive closing parentheses to balance it.
4. **Handling Closing Parentheses ')':**
   - If the current character is a ')', we first check whether it forms a pair with the next character (i.e., if it is followed by another ')'). If yes, we increment `i` by 1 to skip the next character since we have a complete pair '))', and then we decrement `x`.
   - If there is no ')' following the current one, we increment `ans` as we need to insert an additional ')' to have a pair ')'. We then check `x`. If `x` is zero (meaning we have an excess of closing parentheses), we increment `ans` again to insert a '(' before the existing ')'. Otherwise, if `x` is not zero, we simply decrement `x`.
5. **Handle Unmatched Opening Parentheses**: After the loop, if we still have unmatched opening parentheses (i.e., if `x` is greater than zero), it means we need to insert two ')' for each. We do this by adding `x << 1` (which is equivalent to `x * 2`) to `ans`.
6. **Returning the Result**: Finally, the `ans` counter now contains the minimal number of insertions needed to balance the string according to the problem constraints, and we return this value.

The key points in this greedy algorithm are:

- Efficiently keeping track of the number of parentheses we need to insert.
- Balancing insertions only when necessary to fulfill the condition that each '(' is followed by two ')'.
- Using bitwise shift `x << 1` as a quick operation to double the `x` value, which is equivalent to adding two closing parentheses for every unmatched '(' at the end of the iteration.

By following this approach, we can guarantee the minimum number of insertions needed to achieve a balanced string.

Example Walkthrough

Consider the string `s = "(()))("`. We need to go through the string and determine how many insertions are required to make this string balanced according to the rules. Let's walk through the solution step-by-step:

1. Initialize two counters:
   - `ans = 0` (counts the total number of insertions needed)
   - `x = 0` (counts the number of open parentheses '(' that need to be paired with closing ones '))')
2. Iterating over the string:
   - Index `i = 0`, character `s[i] = '('`:
     - We increment `x` because this opening parenthesis needs two closing ones.
     - State after this step: `ans = 0`, `x = 1`
   - Index `i = 1`, character `s[i] = '('`:
     - We increment `x` again for another unmatched opening parenthesis.
     - State after this step: `ans = 0`, `x = 2`
   - Index `i = 2`, character `s[i] = ')'`:
     - This right parenthesis could be paired with one from the previous two, but since it should be paired with two, we need another right parenthesis.
     - We increment `ans` to insert one ')' character and decrement `x`.
     - State after this step: `ans = 1`, `x = 1`
   - Index `i = 3`, character `s[i] = ')'`:
     - This right parenthesis forms a valid pair with the previous '('.
     - We decrement `x`.
     - State after this step: `ans = 1`, `x = 0`
   - Index `i = 4`, character `s[i] = ')'`:
     - Since `x` is zero, this right parenthesis is extra and lacks a corresponding '('.
     - We increment `ans` to add an opening '(' before it.
     - State after this step: `ans = 2`, `x = 0`
   - Index `i = 5`, character `s[i] = '('`:
     - We have an opening parenthesis that needs two closing ones.
     - Increment `x`.
     - State after this step: `ans = 2`, `x = 1`
3. Handle unmatched opening parentheses:
   - The iteration ends with `x` greater than zero; hence, we need to insert two ')' for the unmatched '('.
   - Update `ans` by `x << 1` which is equivalent to adding two ')' characters for the last '('.
   - `ans = ans + (x * 2) = 2 + (1 * 2) = 4`
4. Returning the result:
   - With `ans` equalling to 4, we conclude that four insertions are needed to make the string `"(()))("` balanced according to the given rules.

The final balanced string after insertions would look like `"(()))()()"`.

Python Solution

```python
class Solution:
    def minInsertions(self, s: str) -> int:
        # 'balance' keeps track of the balance of the parentheses
        # 'insertions_needed' will be the answer, representing the minimum insertions needed
        insertions_needed = balance = 0
        i, n = 0, len(s) # 'i' is the current position, 'n' is the length of the string

        while i < n:  # Iterate through the string
            if s[i] == '(':  # If the current character is an opening parenthesis
                balance += 1  # Increase balance
            else:  # If the current character is a closing parenthesis
                # Check if there's a consecutive closing parenthesis
                if i + 1 < n and s[i + 1] == ')':
                    i += 1  # Move to the next character as we've found a pair "))"
                else:
                    # If a pair wasn't found, one insertion is needed
                    insertions_needed += 1
                # If there is no unmatched opening parenthesis
                if balance == 0:
                    # Another insertion for an opening parenthesis
                    insertions_needed += 1
                else:
                    # Otherwise, use one unmatched opening to balance a pair "))"
                    balance -= 1
            i += 1  # Move to the next character

        # After processing the entire string, we might have unmatched opening parentheses
        # Each of these needs two insertions to be balanced (one opening parenthesis needs "))")
        insertions_needed += balance * 2

        return insertions_needed  # Return the total number of insertions needed

```

Java Solution

```java
public class Solution {
  public int minInsertions(String s) {
    // Initialize a counter for the insertions needed and a counter for open parentheses
    int insertionsCount = 0, openParensCount = 0;
    int n = s.length();

    // Iterate through each character in the string
    for (int i = 0; i < n; ++i) {
      char currentChar = s.charAt(i);

      // If we encounter an open parenthesis, we increment the open parentheses count
      if (currentChar == '(') {
        ++openParensCount;
      } else {
        // Check if the next character is also a close parenthesis
        if (i + 1 < n && s.charAt(i + 1) == ')') {
          // If it is, we move the index ahead since this is a valid pair of close parentheses
          ++i;
        } else {
          // If it's not, we need an extra insertion to complete a pair
          ++insertionsCount;
        }

        // If there are no open parentheses to match, we need an insertion for an open parenthesis
        if (openParensCount == 0) {
          ++insertionsCount;
        } else {
          // Otherwise, we found a matching pair, so decrement the open parentheses count
          --openParensCount;
        }
      }
    }

    // After processing all characters, we may have unmatched open parentheses.
    // Each one requires two insertions to form a complete "()()"
    insertionsCount += openParensCount * 2;

    // Return the total number of insertions needed to balance the string
    return insertionsCount;
  }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minInsertions(string s) {
        int additionalInsertionsNeeded = 0; // This will count the insertions needed to balance the string.
        int openBracketsCount = 0; // This will keep track of the count of '(' characters seen.
        int stringLength = s.size(); // Length of the input string.

        for (int i = 0; i < stringLength; ++i) {
            if (s[i] == '(') {
                // If the current character is '(', increment the open brackets count.
                ++openBracketsCount;
            } else {
                // If the current character is ')', check if the next character is also ')'.
                if (i + 1 < stringLength && s[i + 1] == ')') {
                    // If yes, skip the next character as '))' is a valid pair.
                    ++i;
                } else {
                    // If no, one additional insertion is needed as we expect ')'.
                    ++additionalInsertionsNeeded;
                }

                // Now, check if there is an open bracket ('(') available to match the closing bracket.
                if (openBracketsCount == 0) {
                    // If not, insert an additional '(' before the current ')'.
                    ++additionalInsertionsNeeded;
                } else {
                    // If there is an open bracket, pair it with the current closing bracket.
                    --openBracketsCount;
                }
            }
        }

        // After processing all characters,
        // we still might have some '(' brackets open which need to be closed.
        // For each '(', we need to insert '))' to balance.
        additionalInsertionsNeeded += openBracketsCount * 2;

        return additionalInsertionsNeeded; // Return the count of insertions needed.
    }
};
```

Typescript Solution

```typescript
function minInsertions(s: string): number {
    let additionalInsertionsNeeded = 0; // This will count the insertions needed to balance the string.
    let openBracketsCount = 0; // This will keep track of the count of '(' characters seen.
    let stringLength = s.length; // Length of the input string.

    for (let i = 0; i < stringLength; ++i) {
        if (s[i] === '(') {
            // If the current character is '(', increment the open brackets count.
            ++openBracketsCount;
        } else {
            // If the current character is ')', check if the next character is also ')'.
            if (i + 1 < stringLength && s[i + 1] === ')') {
                // If yes, skip the next character as '))' is a valid pair.
                ++i;
            } else {
                // If no, one additional insertion is needed as we expect ')' to make a pair.
                ++additionalInsertionsNeeded;
            }

            // Now, check if there is an open bracket ('(') available to match the current closing bracket.
            if (openBracketsCount === 0) {
                // If not, insert an additional '(' before the current ')' to balance.
                ++additionalInsertionsNeeded;
            } else {
                // If there is an open bracket, pair it with the current closing bracket.
                --openBracketsCount;
            }
        }
    }

    // After processing all characters,
    // if there might still be some '(' brackets open which need to be closed.
    // For each '(', we need to insert '))' to balance.
    additionalInsertionsNeeded += openBracketsCount * 2;

    return additionalInsertionsNeeded; // Return the total count of insertions needed.
}
```

Time and Space Complexity

The given Python code aims to find the minimum number of insertions required to balance a string of parentheses.

Time Complexity:

The time complexity of the function is determined by how it iterates through the input string:

- There's a single loop through the input string, which traverses each character exactly once.
- Inside the loop, operations are constant-time arithmetic, if checks, and a single potential increment of the loop variable `i`.
- There is no nested loop or function calls that depend on the size of input inside the loop.

Considering the length of the string as `n`, the total number of steps is proportional to `n`. Therefore, the time complexity of the code is `O(n)`.

Space Complexity:

The space complexity is determined by the amount of extra space used besides the input itself. In the case of this function:

- `ans`, `x`, `i`, and `n` are simple integer variables that occupy constant space.
- There is no use of any data structures that can grow with the input size.
- There are no recursive calls that would add to the call stack space.

Since the space used does not depend on the input size and remains constant, the space complexity of the code is `O(1)`.