678. Valid Parenthesis String

Medium Stack Greedy String Dynamic Programming Leetcode Link

Problem Description

The problem is to determine if a given string `s` is a valid string based on certain rules. The string `s` consists of only three types of characters: '(', ')' and ''. *The characters must form a valid sequence based on the following criteria: every '(' character must have a corresponding ')' character, every ')' must have a corresponding '(', the '(' must occur before its corresponding ')', and the* ' character can be interpreted as either a '(', a ')', or an empty string. We need to assess whether we can rearrange the '*' characters in such a way that all the parentheses are correctly matched, and if so, we will return true; otherwise, we return false.

Intuition

Approaching the solution involves considering the flexibility that the '' *character affords. The* ' can potentially balance out any unmet parentheses if placed correctly, which is crucial to forming a valid string. To solve the problem, we have to deal with the ambiguity of the '*', determining when it should act as a '(', a ')', or be ignored as an empty string ' "" '.

The algorithm employs two passes to ensure that each parenthesis is paired up. In the first pass, we treat every '' *as '(', increasing our "balance" or counter whenever we encounter a '(' or* ', and decreasing it when we encounter a ')'. If our balance drops to zero, it means we have matched all parentheses thus far. However, if it drops below zero, there are too many ')' characters without a matching '(' or '*', so the string can't be valid.

On the second pass, we reverse the string and now treat every '' *as ')'. We then perform a similar count, increasing the balance when we see a ')' or* ', and decreasing it for '('. If the balance drops below zero this time, it means that there are too many '(' characters without a matching ')' or '*', and the string is again invalid.

If we complete both passes without the balance going negative, it means that for every '(' there is a corresponding ')', and the string is valid. We have effectively managed the ambiguity of '*' by checking that they can serve as a viable substitute for whichever parenthesis is needed to complete a valid set.

Our method ensures that all '(' have a corresponding ')', and vice versa, by treating '*' as placeholder for the correct parenthesis needed, thus validating the input string.

Solution Approach

The reference solution approach suggests using dynamic programming to solve the problem. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is typically used when a problem has overlapping subproblems and a recursive structure that allows solutions to these subproblems to be reused.

However, the provided solution code takes a different, more efficient approach by making two single pass scans of the string: one from left to right, and another from right to left.

In the first pass (left to right), we utilize a counter `x` to track the balance between the '(' and ')' characters. We increment `x` when we encounter a '(' or a '', *and decrement it when we encounter a ')'. If we encounter a ')' but `x` is already `0`, it means there's no '(' or* ' preceding it that can be used to make a valid pair, thus the string is invalid and we return `false`.

```
x = 0
for c in s:
    if c in '(*':
        x += 1
    elif x:
        x -= 1
    else:
        return False
```

In the second pass (right to left), we reset the counter `x` and treat '' *as ')'. We increment `x` for ')' or* ', and decrement it for '('. Similar to the first pass, if we encounter a '(' but `x` is `0`, it means there's no ')' or '*' to pair with it, so the string can't be valid.

```
x = 0
for c in s[::-1]:
    if c in '*)':
        x += 1
    elif x:
        x -= 1
    else:
        return False
```

After both scans, if our balance has never gone negative, we can conclude that for every '(' there is a matching ')' (after considering that '*' could count as either), so the string is valid.

This solution is more efficient than the dynamic programming approach mentioned in the reference solution approach. While dynamic programming would have a time complexity of `O(n^3)` and a space complexity of `O(n^2)`, the provided solution has a time complexity of `O(n)` and a space complexity of `O(1)`, since it simply uses a single integer for tracking and iterates through the string twice, without requiring any additional data structures.

Example Walkthrough

Let's take the string `s = "(*))"` as an example to illustrate the solution approach.

First Pass (Left to Right):

We will initialize our balance counter `x` to `0`. We iterate through the string one character at a time.

1. `s[0] = '('`: Since it is a '(', we increment `x` to `1`.
2. `s[1] = '*'`: Since it is a '*', it could be '(', ')', or empty. We treat it as '(', so we increment `x` to `2`.
3. `s[2] = ')'`: We have a ')', so this could potentially pair with the previous '(' or '*'. We decrement `x` to `1`.
4. `s[3] = ')'`: Another ')'. It can pair with the '(' or '*' we assumed in step 2. We decrement `x` to `0`.

At the end of this pass, `x` is not negative, which means that we have not encountered a ')' that could not be matched with a previous '(' or '*' treated as '('.

Second Pass (Right to Left):

Now, we will reset `x` to `0` and traverse from right to left, treating '*' as ')'.

1. `s[3] = ')'`: We increment `x` to `1` since it could pair with an '(', or a '*', treated as '('.
2. `s[2] = ')'`: We increment `x` to `2`—similar reasoning as step 1.
3. `s[1] = '*'`: Treating it as ')', we increment `x` to `3`.
4. `s[0] = '('`: We have an '(', so we pair it with one of the ')' or '*' we treated as ')'. We decrement `x` to `2`.

After the second pass, `x` is not negative, confirming that we have never encountered a '(' character that could not be matched with a subsequent ')' or '*' treated as ')'.

Since we finished both passes without the balance `x` going negative, we can conclude that the string `s = "(*))"` can be rearranged into a valid sequence. Therefore, our function would return `true` for this input.

Python Solution

```python
class Solution:
    def checkValidString(self, s: str) -> bool:
        # Initialization of the balance counter for open parentheses.
        open_balance = 0

        # Forward iteration over string to check if it can be valid from left to right.
        for char in s:
            # If character is '(' or '*', it could count as a valid open parenthesis,
            # so increase the balance counter.
            if char in '(*':
                open_balance += 1
            # If the character is ')', decrease the balance counter as a ')' is closing
            # an open parenthesis.
            elif open_balance:
                open_balance -= 1
            # If there's no open parenthesis to balance the closing one, return False.
            else:
                return False

        # Reinitialization of the balance counter for closed parentheses.
        closed_balance = 0

        # Backward iteration over string to check if it can be valid from right to left.
        for char in s[::-1]:
            # If character is ')' or '*', it could count as a valid closed parenthesis,
            # so increase the closed_balance counter.
            if char in '*)':
                closed_balance += 1
            # If the character is '(', decrease the closed_balance counter as '(' is
            # potentially closing a prior ')'.
            elif closed_balance:
                closed_balance -= 1
            # If there's no closing parenthesis to balance the opening one, return False.
            else:
                return False

        # If all parentheses and asterisks can be balanced in both directions,
        # the string is considered valid.
        return True

```

Java Solution

```java
class Solution {

    // Method to check if a string with parentheses and asterisks (*) is valid
    public boolean checkValidString(String s) {
        int balance = 0; // This will keep track of the balance of open parentheses
        int n = s.length(); // Length of the input string

        // First pass goes from left to right
        for (int i = 0; i < n; ++i) {
            char currentChar = s.charAt(i);
            if (currentChar != ')') {
                // Increment balance for '(' or '*'
                ++balance;
            } else if (balance > 0) {
                // Decrement balance if there is an unmatched '(' before
                --balance;
            } else {
                // A closing ')' without a matching '('
                return false;
            }
        }

        // Reset balance for the second pass
        balance = 0;

        // Second pass goes from right to left
        for (int i = n - 1; i >= 0; --i) {
            char currentChar = s.charAt(i);
            if (currentChar != '(') {
                // Increment balance for ')' or '*'
                ++balance;
            } else if (balance > 0) {
                // Decrement balance if there is an unmatched ')' after
                --balance;
            } else {
                // An opening '(' without a matching ')'
                return false;
            }
        }

        // If we did not return false so far, the string is valid
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if the string with parentheses and asterisks is valid
    bool checkValidString(string s) {
        int balance = 0; // Track the balance of the parentheses
        int n = s.size(); // Store the size of the string

        // Forward pass to ensure there aren't too many closing parentheses
        for (int i = 0; i < n; ++i) {
            // Increment balance for an opening parenthesis or an asterisk
            if (s[i] != ')') {
                ++balance;
            }
            // Decrement balance for a closing parenthesis if balance is positive
            else if (balance > 0) {
                --balance;
            }
            // If balance is zero, too many closing parentheses are encountered
            else {
                return false;
            }
        }

        // If only counting opening parentheses and asterisks, balance might be positive
        // So we check in the reverse order for the opposite scenario
        balance = 0; // Reset balance for the backward pass
        for (int i = n - 1; i >= 0; --i) {
            // Increment balance for closing parenthesis or an asterisk
            if (s[i] != '(') {
                ++balance;
            }
            // Decrement balance for an opening parenthesis if balance is positive
            else if (balance > 0) {
                --balance;
            }
            // If balance is zero, too many opening parentheses are encountered
            else {
                return false;
            }
        }
        // If the string passes both forward and backward checks, it's valid
        return true;
    }
};

```

Typescript Solution

```typescript
// Global variable to track the balance of parentheses
let balance: number;

// Global variable to store the size of the string
let n: number;

// Function to check if the string with parentheses and asterisks is valid
function checkValidString(s: string): boolean {
    balance = 0; // Initialize balance
    n = s.length; // Get the length of the string

    // Forward pass to ensure there aren't too many closing parentheses
    for (let i = 0; i < n; ++i) {
        if (s[i] !== ')') {
            // Increment balance for an opening parenthesis or an asterisk
            balance++;
        } else if (balance > 0) {
            // Decrement balance for a closing parenthesis if balance is positive
            balance--;
        } else {
            // If balance is zero, too many closing parentheses are encountered
            return false;
        }
    }

    if (balance === 0) {
        // If balance is zero, it means we have exact matches, so return true
        return true;
    }

    // Reset balance for the backward pass
    balance = 0;
    for (let i = n - 1; i >= 0; --i) {
        if (s[i] !== '(') {
            // Increment balance for closing parenthesis or an asterisk
            balance++;
        } else if (balance > 0) {
            // Decrement balance for an opening parenthesis if balance is positive
            balance--;
        } else {
            // If balance is zero, too many opening parentheses are encountered
            return false;
        }
    }

    // If the string passes both forward and backward checks, it's valid
    return true;
}

```

Time and Space Complexity

Time Complexity

The provided algorithm consists of two for-loops that scan through the string `s`. Each loop runs independently from the beginning and from the end of the string, checking conditions and updating the variable `x` accordingly.

The time complexity of each pass is `O(n)`, where `n` is the length of string `s`, since each character is examined exactly once in each pass.

Since there are two passes through the string, the total time complexity of the algorithm is `O(n) + O(n)` which simplifies to `O(n)`.

Space Complexity

The space complexity of the algorithm depends on the additional space used by the algorithm, excluding the input itself.

In this case, only a single integer `x` is used to keep track of the balance of parentheses and asterisks, which occupies constant space.

Hence, the space complexity of the algorithm is `O(1)`, as it does not depend on the size of the input string `s` and only uses a fixed amount of additional space.