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`.

```
1 x = 0
2 for c in s:
3     if c in '(*':
4         x += 1
5     elif x:
6         x -= 1
7     else:
8         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.

```
1 x = 0
2 for c in s[::-1]:
3     if c in '*)':
4         x += 1
5     elif x:
6         x -= 1
7     else:
8         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

```
1  class Solution:
2      def checkValidString(self, s: str) -> bool:
3          # Initialization of the balance counter for open parentheses.
4          open_balance = 0
5
6          # Forward iteration over string to check if it can be valid from left to right.
7          for char in s:
8              # If character is '(' or '*', it could count as a valid open parenthesis,
9              # so increase the balance counter.
10             if char in '(*':
11                 open_balance += 1
12             # If the character is ')', decrease the balance counter as a ')' is closing
13             # an open parenthesis.
14             elif open_balance:
15                 open_balance -= 1
16             # If there's no open parenthesis to balance the closing one, return False.
17             else:
18                 return False
19
20         # Reinitialization of the balance counter for closed parentheses.
21         closed_balance = 0
22
23         # Backward iteration over string to check if it can be valid from right to left.
24         for char in s[::-1]:
25             # If character is ')' or '*', it could count as a valid closed parenthesis,
26             # so increase the closed_balance counter.
27             if char in '*)':
28                 closed_balance += 1
29             # If the character is '(', decrease the closed_balance counter as '(' is
30             # potentially closing a prior ')'.
31             elif closed_balance:
32                 closed_balance -= 1
33             # If there's no closing parenthesis to balance the opening one, return False.
34             else:
35                 return False
36
37         # If all parentheses and asterisks can be balanced in both directions,
38         # the string is considered valid.
39         return True
40
```

Java Solution

```
1  class Solution {
2
3      // Method to check if a string with parentheses and asterisks (*) is valid
4      public boolean checkValidString(String s) {
5          int balance = 0; // This will keep track of the balance of open parentheses
6          int n = s.length(); // Length of the input string
7
8          // First pass goes from left to right
9          for (int i = 0; i < n; ++i) {
10             char currentChar = s.charAt(i);
11             if (currentChar != '*') {
12                 // Increment balance for '(' or '*'
13                 ++balance;
14             } else if (balance > 0) {
15                 // Decrement balance if there is an unmatched '(' before
16                 --balance;
17             } else {
18                 // A closing ')' without a matching '('
19                 return false;
20             }
21         }
22
23         // Reset balance for the second pass
24         balance = 0;
25
26         // Second pass goes from right to left
27         for (int i = n - 1; i >= 0; --i) {
28             char currentChar = s.charAt(i);
29             if (currentChar != '(') {
30                 // Increment balance for ')' or '*'
31                 ++balance;
32             } else if (balance > 0) {
33                 // Decrement balance if there is an unmatched ')' after
34                 --balance;
35             } else {
36                 // An opening '(' without a matching ')'
37                 return false;
38             }
39         }
40
41         // If we did not return false so far, the string is valid
42         return true;
43     }
44 }
45
```

C++ Solution

```
1  class Solution {
2  public:
3      // Function to check if the string with parentheses and asterisks is valid
4      bool checkValidString(string s) {
5          int balance = 0; // Track the balance of the parentheses
6          int n = s.size(); // Store the size of the string
7
8          // Forward pass to ensure there aren't too many closing parentheses
9          for (int i = 0; i < n; ++i) {
10             // Increment balance for an opening parenthesis or an asterisk
11             if (s[i] != ')') {
12                 ++balance;
13             }
14             // Decrement balance for a closing parenthesis if balance is positive
15             else if (balance > 0) {
16                 --balance;
17             }
18             // If balance is zero, too many closing parentheses are encountered
19             else {
20                 return false;
21             }
22         }
23
24         // If only counting opening parentheses and asterisks, balance might be positive
25         // So we check in the reverse order for the opposite scenario
26         balance = 0; // Reset balance for the backward pass
27         for (int i = n - 1; i >= 0; --i) {
28             // Increment balance for closing parenthesis or an asterisk
29             if (s[i] != '(') {
30                 ++balance;
31             }
32             // Decrement balance for an opening parenthesis if balance is positive
33             else if (balance > 0) {
34                 --balance;
35             }
36             // If balance is zero, too many opening parentheses are encountered
37             else {
38                 return false;
39             }
40         }
41         // If the string passes both forward and backward checks, it's valid
42         return true;
43     }
44 };
45
```

Typescript Solution

```
1  // Global variable to track the balance of parentheses
2  let balance: number;
3
4  // Global variable to store the size of the string
5  let n: number;
6
7  // Function to check if the string with parentheses and asterisks is valid
8  function checkValidString(s: string): boolean {
9      balance = 0; // Initialize balance
10     n = s.length; // Get the length of the string
11
12     // Forward pass to ensure there aren't too many closing parentheses
13     for (let i = 0; i < n; ++i) {
14         if (s[i] !== ')') {
15             // Increment balance for an opening parenthesis or an asterisk
16             ++balance;
17         } else if (balance > 0) {
18             // Decrement balance for a closing parenthesis if balance is positive
19             --balance;
20         } else {
21             // If balance is zero, too many closing parentheses are encountered
22             return false;
23         }
24     }
25
26     if (balance === 0) {
27         // If balance is zero, it means we have exact matches, so return true
28         return true;
29     }
30
31     // Reset balance for the backward pass
32     balance = 0;
33     for (let i = n - 1; i >= 0; --i) {
34         if (s[i] !== '(') {
35             // Increment balance for closing parenthesis or an asterisk
36             ++balance;
37         } else if (balance > 0) {
38             // Decrement balance for an opening parenthesis if balance is positive
39             --balance;
40         } else {
41             // If balance is zero, too many opening parentheses are encountered
42             return false;
43         }
44     }
45
46     // If the string passes both forward and backward checks, it's valid
47     return true;
48 }
49
```

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.