12.01.2025

Problem: Check if a Parentheses String Can Be Valid

Problem Statement: You are given a parentheses string `s` and a binary string `locked`, both of length `n`. The `locked` string tells you whether you can change the corresponding character in `s`:

- If `locked[i] == '1'`, the character at index `i` in `s` is fixed.
- If `locked[i] == '0'`, you can change the character at index `i` in `s` to either `(` or `)`.

Your task is to determine if it's possible to change some characters in `s` (where `locked[i] == '0'`) such that the string becomes a valid parentheses string.

Approach:

There are two primary methods to solve this problem:

---

1. Using Two Counters (Efficient Solution)

The idea is to perform two passes through the string:

1. **Left-to-right pass**: Count how many open parentheses (`(`) and close parentheses (`)`) are encountered, ensuring the string doesn't become invalid at any point.
2. **Right-to-left pass**: Ensure that after the first pass, the parentheses can be balanced from the other direction as well.

Steps:

1. Traverse the string from left to right:
   a. If `s[i]` is `(` or `locked[i] == '0'` (indicating we can make it a `(`), increment the count of open parentheses.
   b. If `s[i]` is `)` and `locked[i] == '1'`, decrement the count of open parentheses.
   c. If at any point the count of open parentheses goes negative, it means the parentheses are imbalanced, and we return `false`.
2. Traverse the string from right to left:
   a. Similar to the left-to-right pass, check if it's possible to balance the parentheses in reverse order.
3. Return `true` if both passes are successful, otherwise return `false`.

Code:

cpp

Copy code

```
class Solution { public: bool canBeValid(string s, string locked) { int n = s.length();
if (n % 2 != 0) { return false; // If the length is odd, it can't be valid } // Left to
right pass: ensure balanced open parentheses int countOpen = 0; for (int i = 0; i < n;
i++) { if (s[i] == '(' || locked[i] == '0') { countOpen++; // Can make it an opening
parenthesis } else { countOpen--; // It's a closing parenthesis } if (countOpen < 0)
return false; // Too many closing parentheses } // Right to left pass: ensure balanced
closing parentheses int countClose = 0; for (int i = n - 1; i >= 0; i--) { if (s[i] ==
')' || locked[i] == '0') { countClose++; // Can make it a closing parenthesis } else {
countClose--; // It's an opening parenthesis } if (countClose < 0) return false; // Too
many opening parentheses } return true; // Valid if both passes succeed } };
```

Time Complexity:

- **O(n)**: We traverse the string twice, making this approach linear in time.

Space Complexity:

- **O(1)**: Only a few variables are used, making the space complexity constant.