12.01.2025 - Check if a Parentheses String Can Be Valid

Valid Parentheses String with Locked Positions

Problem Statement

Given:

- A string `s` containing parentheses.
- A binary string `locked` where:
  a. `locked[i] == '1'` means the parenthesis at position `i` is locked and cannot be changed.
  b. `locked[i] == '0'` means the parenthesis at position `i` can be changed (`(` to `)` or vice versa).

Return `true` if the string can be converted into a valid parentheses string, else return `false`.

Valid Parentheses Examples:

- `s = "(())"` is valid.
- `s = "((()))"` is valid.
- `s = "()()()"` is valid.

Observations:

1. **Odd Length Check:** A string with odd length can never be a valid parentheses string.
2. **Stack Consideration:** When dealing with parentheses, a stack-based approach is often effective.
3. **Locked and Unlocked Handling:** If a parenthesis is unlocked, it can be swapped.
4. **Closing Bracket Position:** A closing bracket should always have an opening bracket before it.

Approach Explanation:

We will use a modified stack-based algorithm to validate the parentheses with the following steps:

1. **Stack Choice:** Use two stacks:
   a. `openStack`: Stores indices of locked opening brackets `(`.
   b. `openCloseStack`: Stores indices of unlocked brackets (can be changed if necessary).
2. **Iteration Logic:**
   a. If `locked[i] == '0'` (unlocked), add the index to `openCloseStack`.
   b. If `locked[i] == '1'` and `s[i] == '('`, add the index to `openStack`.
   c. If `locked[i] == '1'` and `s[i] == ')'`:
      i. Try to balance it with `openStack`. If not possible,
      ii. Try to balance it with `openCloseStack`.

iii. If both stacks are empty, return `false`.

3. **Post Iteration Check:**
   a. Try to balance the remaining open brackets with unlocked indices.
   b. If unbalanced brackets remain, return `false`.

Code Implementation (C++):

```cpp
class Solution {
public:
    bool canBeValid(string s, string locked) {
        int n = s.length();
        if (n % 2 != 0) return false; // Odd length check

        stack<int> openStack;        // Locked opening brackets
        stack<int> openCloseStack;  // Unlocked brackets

        // Pass through the string
        for (int i = 0; i < n; i++) {
            if (locked[i] == '0') {
                openCloseStack.push(i); // Unlocked bracket
            } else if (s[i] == '(') {
                openStack.push(i);      // Locked open bracket
            } else if (s[i] == ')') {
                if (!openStack.empty()) {
                    openStack.pop();     // Balance with locked open bracket
                } else if (!openCloseStack.empty()) {
                    openCloseStack.pop(); // Balance with unlocked bracket
                } else {
                    return false;         // No way to balance
                }
            }
        }

        // Attempt to balance any remaining open brackets
        while (!openStack.empty() && !openCloseStack.empty() && openStack.top() <
openCloseStack.top()) {
            openStack.pop();
            openCloseStack.pop();
        }

        return openStack.empty();
    }
};
```

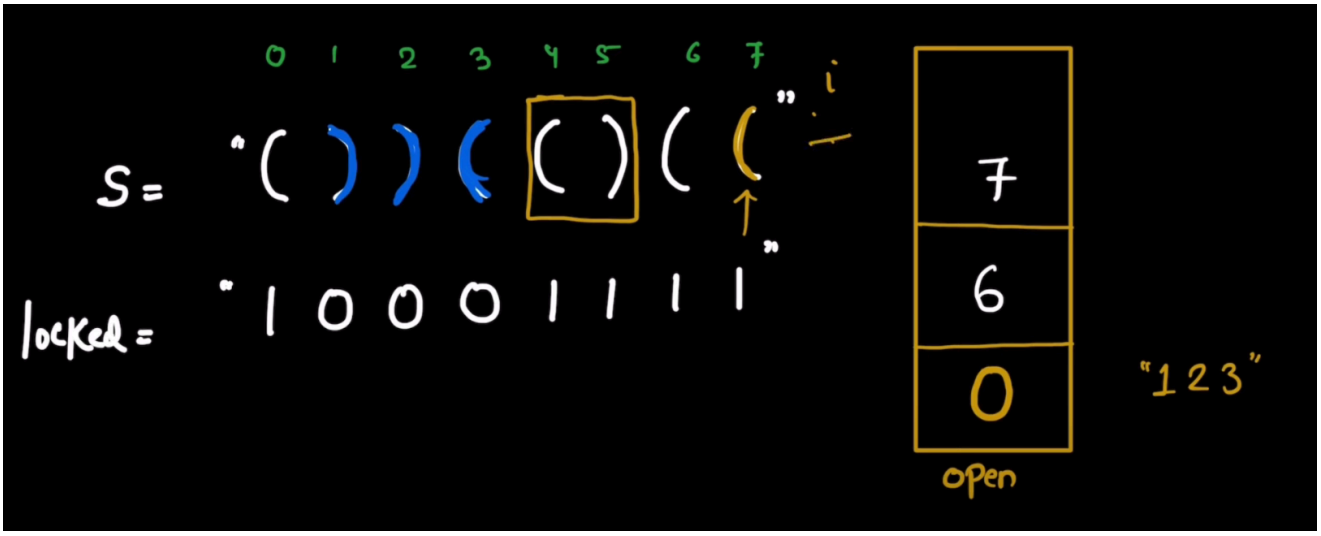
Complexity:

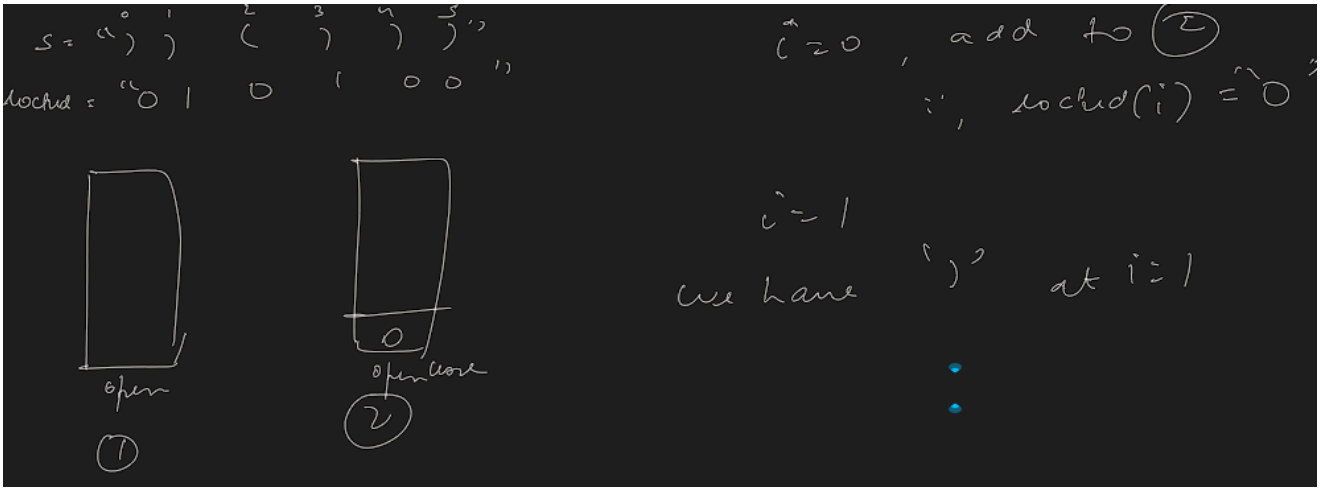
- **Time Complexity:** `O(n)` – Each index is processed once.
- **Space Complexity:** `O(n)` – Space for two stacks.

Edge Cases:

- If the length of `s` is odd, return `false` immediately.
- If there are no locked parentheses, the string can always be balanced.

This approach ensures an efficient and optimal solution to the problem.



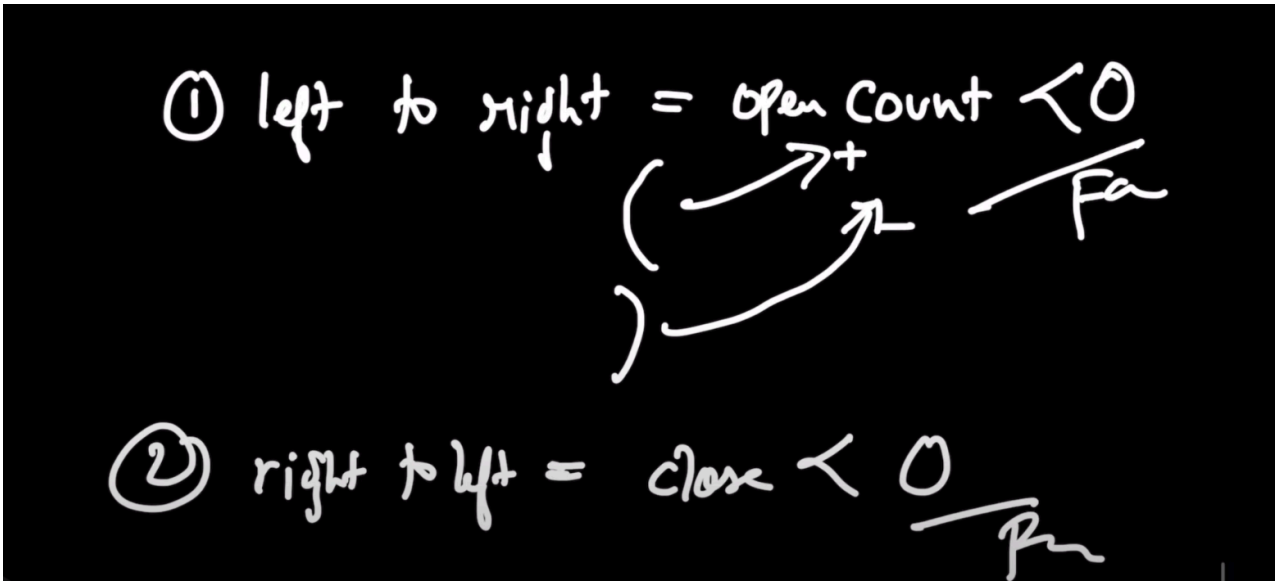


Approach 2

for checking if the given string is valid string or not without any locked string

1. go from left to right: (maintain opening bracket count)
    a. we can maintain the count of open and close bracket in one variable only i.e. if we encounter opening bracket, we increment by 1 and if we encounter a closing bracket, we decrement by 1.
    b. if count is less than 0 then we can say that we have exceeded the number of closing brackets, therefore any further closing bracket will not be able to make it a valid pair as any bracket ahead of that bracket that made the count as negative would be creating a pair with the closing brackets on their right.
2. go from right to left: (maintain closing bracket count)
    a. when we encounter opening bracket decrement the count 1 and we encounter an closing bracket increment the count 1.

b. as count become negative, return false.

3. 

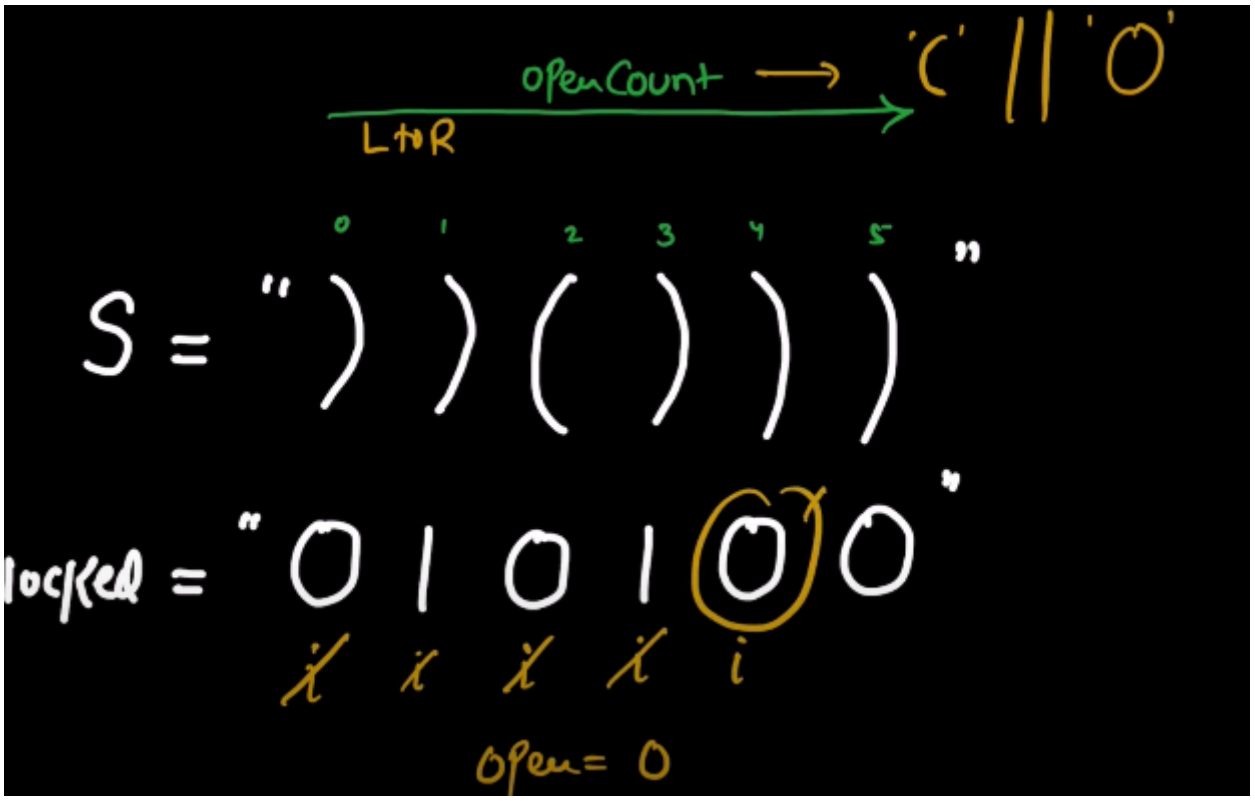
now comes the part of using locked string

Left to Right Approach:

During the left-to-right traversal, the primary modification involves how the count is updated. Instead of only incrementing the count for an opening bracket `'('`, we will also increment the count when the corresponding `locked[i] == '0'`, as it indicates the bracket can be adjusted if needed.

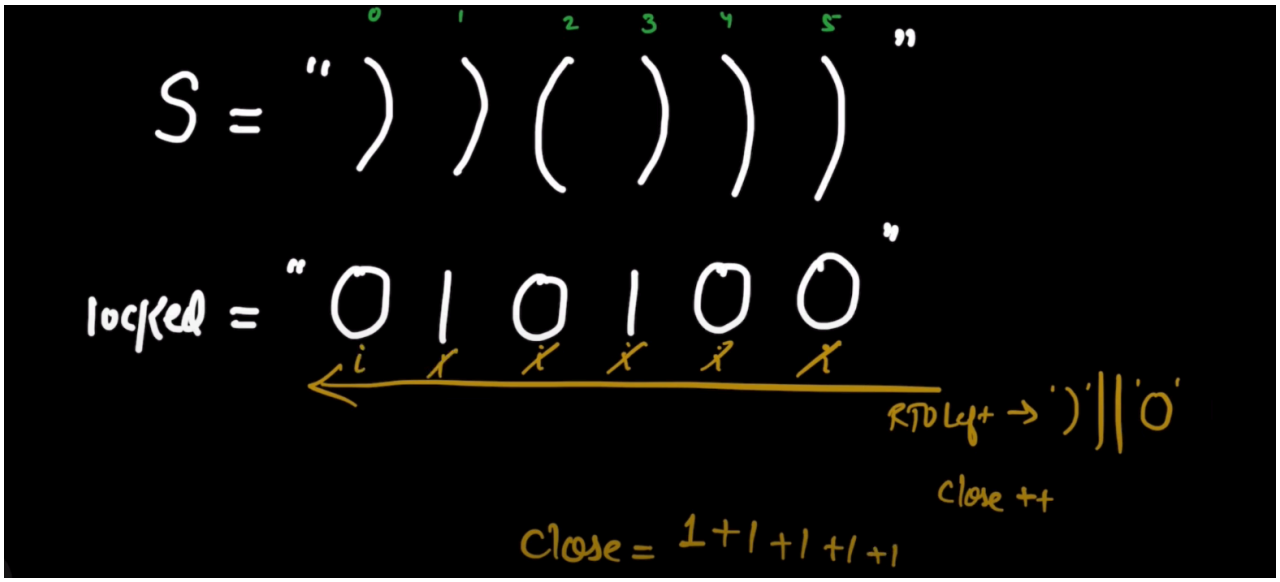
- **Increment the count** when encountering an opening bracket `'('` or an unlocked position (`locked[i] == '0'`).
- **Decrement the count** when encountering a closing bracket `')'` or a locked closing bracket (`locked[i] == '1'`).

This ensures flexibility in balancing the parentheses string while maintaining the conditions for a valid sequence.



right to left

1. if closing bracket, increase the close count or if we have zero then we would increment the count.

2. 


```
class Solution {
public:
    bool canBeValid(string s, string locked) {
        // using 2 variables and 2 traversals.
        int n = s.length();
        if (n % 2 != 0)
            return false;
        // left to right => open count;
```

```
        int countOpen = 0;
        for(int i = 0; i < n; i++){
            if(s[i] == '(' || locked[i] == '0'){
                countOpen++;
            }else{
                countOpen--;
            }
            if(countOpen < 0) return false;
        }
        int countClose = 0;
        for(int i = n- 1; i >=0; i--){
            if(s[i] == ')' || locked[i] == '0'){
                countClose++;
            }else{
                countClose--;
            }
            if(countClose < 0) return false;
        }
        return true;
    }
    // time complexity: O(2 * N)
    // space complexity: O(1)
};
```