2116. Check if a Parentheses String Can Be Valid

Medium Stack Greedy String Leetcode Link

Problem Description

The given problem involves determining whether a string of parentheses `s` can be transformed into a valid parentheses string given certain constraints. A valid parentheses string is defined as follows:

1. It is a single pair of parentheses, i.e., "()".
2. It can be decomposed into two valid parentheses strings, `A` and `B`, arranged side-by-side as `AB`.
3. It can be a valid parentheses string `A` enclosed within a pair of parentheses, like `(A)`.

In addition to the string `s`, there is a binary string `locked`, with the same length as `s`. Each position `i` in `locked` can either be a `'1'` or a `'0'`. If `locked[i]` is `'1'`, the corresponding character `s[i]` is fixed and cannot be changed. However, if `locked[i]` is `'0'`, then `s[i]` can be converted to either `'('` or `')'`.

The goal is to determine if it's possible to make `s` a valid parentheses string by potentially changing some of the parentheses at positions where `locked[i]` is `'0'`.

Intuition

To solve this problem, the solution employs a two-pass scan (one from left to right, and another from right to left) to count if there could be enough parentheses to balance each other out while respecting the constraints imposed by the `locked` string.

The first for loop goes from left to right. It tries to validate that there are enough left parentheses or flexible positions (where `locked[i] == '0'`) to match with potential right parentheses. Whenever a '`(`' or a flexible position is encountered, it is counted as a potential left parenthesis (increment `x`). Whenever a '`)`' is encountered in a non-flexible position, it is matched with a potential left parenthesis (decrement `x`). If at any point `x` becomes negative, it implies there are more right parentheses than can be matched by left ones, violating the balance required for a valid parentheses string.

The left to right pass ensures that at each point from left to right, there are never more right parentheses than left/flexible ones to match them.

The second for loop goes from right to left. It performs the same check but in the other direction, confirming that there are enough right parentheses or flexible positions to match with all the left parentheses.

The scan from right to left validates that there are never more left parentheses than right/flexible ones to match them from right to left.

By using two passes, we ensure the balance of parentheses is maintained in both directions. If both passes succeed without the balance going negative at any point, it guarantees that we can rearrange the parentheses (for positions where `locked[i] == '0'`) to form a valid parentheses string.

The solution also includes an initial check for strings with an odd length, as they cannot possibly form valid parentheses pairings, and such cases return `False` immediately.

Solution Approach

The implementation of the solution uses a single counter variable `x` to keep track of the balance of parentheses during the two-pass scan process. The solution does not require complex data structures or patterns. It relies on simple iteration and condition evaluation. Here's the breakdown of the solution approach:

1. **Initial Length Check**: Before any scanning, we check if the length of the string `s` is odd. Since valid parentheses strings must have an even length (each '`(`' must have a matching '`)`'), an odd length string can never be valid. If the length is odd, we return `False`.

```
n = len(s)
if n & 1:
    return False
```

2. **Left to Right Scan**:
   - Loop through the string from the first character to the last. For each character, check if it's a left parenthesis '`(`' or a flexible position (where `locked[i] == '0'`). In either case, this could potentially be a left parenthesis, so increment the counter `x`.
   - If the character is a right parenthesis '`)`' and is locked, check if there is a previously counted left parentheses to match with it by checking if `x` is greater than `0`. If yes, decrement `x`. If `x` is `0`, it means there are too many right parentheses, so the string cannot be valid.

```
x = 0
for i in range(n):
    if s[i] == '(' or locked[i] == '0':
        x += 1
    elif x:
        x -= 1
    else:
        return False
```

3. **Right to Left Scan**:
   - Now, we do the reverse. Loop through the string from the last character to the first. If we encounter a right parenthesis '`)`' or a flexible position, it could potentially be a right parenthesis, so increment `x`.
   - If the character is a left parenthesis '`(`' and is locked, check if there is a right parenthesis available to pair with it. If `x` is greater than `0`, decrement `x`. If `x` is `0`, it again means an imbalance, and the string cannot be valid.

```
x = 0
for i in range(n - 1, -1, -1):
    if s[i] == ')' or locked[i] == '0':
        x += 1
    elif x:
        x -= 1
    else:
        return False
```

4. **Final Validation**: If neither the left-to-right scan nor the right-to-left scan has returned `False`, the parentheses can potentially be balanced by substituting '(`' or ')' at the flexible positions (`'0'in`locked`). Hence, the function returns` True`.

By methodically scanning the input from both directions, the algorithm ensures the string maintains a balance of parentheses that aligns with the rules of a valid string, considering the constraints given by `locked`. This two-pass approach is a common pattern for validating parentheses strings, often used because it efficiently verifies the balance of parentheses without needing additional storage or complex operations.

Example Walkthrough

Let's walk through an example to illustrate the solution approach with a given string `s` and a corresponding `locked` string.

Suppose we are given the string `s = "))(("` and the `locked` string `locked = "1001"`. The goal is to determine whether we can make `s` a valid parentheses string.

Before we begin with the two scans, we notice that the length of `s` is even, which means it's possible (in terms of length) for `s` to be a valid parentheses string. The initial length check passes.

Left to Right Scan:

- Starting with `x = 0`, we iterate from left to right.
- `s[0] = ')'` and `locked[0] = '1'`. Because `s[0]` is a right parenthesis and is locked, we cannot change it and there's no left parenthesis to match with it (`x = 0`). Hence, this immediately makes the string invalid and the function would return `False`.

In this example, the left-to-right scan fails at the first character itself since it encounters a locked right parenthesis with no possible left parenthesis to pair with. Hence, we do not need to move forward with the right-to-left scan, and we can conclude that it's not possible to transform `s` into a valid parentheses string with the given `locked` constraints.

Python Solution

```python
class Solution:
    def canBeValid(self, s: str, locked: str) -> bool:
        # The length of the string s.
        string_length = len(s)

        # A valid string must have an even number of characters.
        if string_length % 2 != 0:
            return False

        # Counter for unbalanced open parentheses.
        balance = 0

        # First pass to check if we can balance from left to right.
        for index in range(string_length):
            # If we encounter an open parenthesis or an unlocked character, we increment balance.
            if s[index] == '(' or locked[index] == '0':
                balance += 1
            # If we can match an open parenthesis, we decrement balance.
            elif balance:
                balance -= 1
            # If balance is 0 and we encounter a closed parenthesis that is locked, it is unbalanced and can't be valid.
            else:
                return False

        # If after the first pass, we have managed to match every parenthesis, the string may be valid.
        # However, we need to check the same from right to left.

        # Reset balance for the second pass.
        balance = 0

        # Second pass to check if we can balance from right to left.
        for index in range(string_length - 1, -1, -1):
            # If we encounter a closed parenthesis or an unlocked character, we increment balance.
            if s[index] == ')' or locked[index] == '0':
                balance += 1
            # If we can match a closed parenthesis, we decrement balance.
            elif balance:
                balance -= 1
            # If balance is 0 and we encounter an open parenthesis that is locked, it is unbalanced and can't be valid.
            else:
                return False

        # If both passes are successful, the string is valid.
        return True

```

Java Solution

```java
class Solution {

    // Method to determine if a string can be made valid by swapping parentheses
    public boolean canBeValid(String s, String locked) {
        int n = s.length();

        // If the length of the string is odd, it can't be a valid string of parentheses
        if (n % 2 == 1) {
            return false;
        }

        // Initialize the balance counter for left-to-right scan
        int balance = 0;

        // Left-to-right scan to ensure all ')' can be matched
        for (int i = 0; i < n; ++i) {
            // If we have an unlocked character or a '(', we can potentially match a ')'
            if (s.charAt(i) == '(' || locked.charAt(i) == '0') {
                ++balance;
            } else if (balance > 0) {
                // We have a locked ')' and we have an earlier '(', so pair them
                --balance;
            } else {
                // Mismatched ')' which cannot be matched
                return false;
            }
        }

        // Reset the balance counter for right-to-left scan
        balance = 0;

        // Right-to-left scan to ensure all '(' can be matched
        for (int i = n - 1; i >= 0; --i) {
            // If we have an unlocked character or a ')', we can potentially match a '('
            if (s.charAt(i) == ')' || locked.charAt(i) == '0') {
                ++balance;
            } else if (balance > 0) {
                // We have a locked '(' and we have an earlier ')', so pair them
                --balance;
            } else {
                // Mismatched '(' which cannot be matched
                return false;
            }
        }

        // If all the checks pass, the string can be made valid
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine if the string can be made valid by swapping brackets
    bool canBeValid(string s, string locked) {
        int n = s.length(); // n is the length of the input string

        // If the length of the string is odd, it is not possible to have a valid set of brackets
        if (n % 2 != 0) {
            return false;
        }

        // Forward scan: from the beginning to check if we can make a valid string
        int openBrackets = 0; // Track the balance of open brackets
        for (int i = 0; i < n; ++i) {
            // If the current character is an open bracket or it is unlocked, increase the count of open brackets
            // Unlocked characters '0' are treated as wildcards that can be either open or closed brackets
            if (s[i] == '(' || locked[i] == '0') {
                ++openBrackets;
            }
            else if (openBrackets > 0) { // If it is a closed bracket and there is an open bracket available, balance it out
                --openBrackets;
            }
            else {
                return false; // If there is no open bracket available to balance the close, it is invalid
            }
        }

        // Backward scan: from the end to check if we can make a valid string
        int closeBrackets = 0; // Track the balance of close brackets
        for (int i = n - 1; i >= 0; --i) {
            // If the current character is a closed bracket or it is unlocked, treat it similarly to the forward scan
            if (s[i] == ')' || locked[i] == '0') {
                ++closeBrackets;
            }
            else if (closeBrackets > 0) { // If there is a close bracket available to balance the open, balance it out
                --closeBrackets;
            }
            else {
               return false; // If there is no close bracket available to balance the open bracket, it is invalid
            }
        }

        // If both scans do not detect an imbalance, the string can be made valid
        return true;
    }
};

```

Typescript Solution

```typescript
// Function to determine if the string can be made valid by swapping brackets
function canBeValid(s: string, locked: string): boolean {
    let n: number = s.length; // n is the length of the input string

    // If the length of the string is odd, it is impossible to form a valid string of brackets
    if (n % 2 !== 0) {
        return false;
    }

    // Forward scan: from the beginning to check if we can make a valid string
    let openBrackets: number = 0; // Track the balance of open brackets
    for (let i: number = 0; i < n; ++i) {
        // If the current character is an open bracket or it is unlocked ('0'), increment open brackets
        if (s[i] === '(' || locked[i] === '0') {
            ++openBrackets;
        } else if (openBrackets > 0) { // If a closing bracket is encountered and there is an open bracket to pair with, decrement
            --openBrackets;
        } else {
            return false; // If there are no open brackets to pair with a closing one, it's invalid
        }
    }

    // Backward scan: from the end to check if we can make a valid string
    let closeBrackets: number = 0; // Track the balance of close brackets
    for (let i: number = n - 1; i >= 0; --i) {
        // If the current character is a closed bracket or unlocked ('0'), increment close brackets
        if (s[i] === ')' || locked[i] === '0') {
            ++closeBrackets;
        } else if (closeBrackets > 0) { // If an opening bracket is encountered and there is a close bracket to pair with, decremen
            --closeBrackets;
        } else {
            return false; // If there are no close brackets to pair with an opening one, it's invalid
        }
    }

    // If both scans do not detect an imbalance, the string can be made valid
    return true;
}

```

Time and Space Complexity

The given Python code defines a method `canBeValid` which determines if a string `s` of parentheses can be a valid expression by toggling some locked/unlocked characters defined by `locked`, where '0' represents an unlocked character and '1' represents a locked character.

Time Complexity

The time complexity of the function is `O(n)`, where `n` is the length of the string `s`. This complexity arises from the fact that there are two separate for-loops that each iterate through the length of the string once.

1. The first loop starts from index 0 and goes up to `n - 1`.
2. The second loop starts from index `n - 1` and goes down to 0.

Neither of the loops is nested, and each pass through the entire string consists of `O(1)` operations within the loop, leading to a linear time complexity in terms of the length of the string for both passes.

Therefore, the overall time complexity of the method `canBeValid` is `O(n) + O(n)`, which simplifies to `O(n)`.

Space Complexity

The space complexity of the function is `O(1)`. The only extra space used is for simple integer variables that are used for loop iteration and keeping track of the balance of parentheses. The space used does not scale with the size of the input string `s`, as there are no data structures dependent on `n` used for storage. All operations are done in-place with a fixed amount of extra storage space, therefore the space complexity is constant.