1614. Maximum Nesting Depth of the Parentheses

Easy Stack String

Problem Description

The problem defines a **valid parentheses string** (VPS) with specific rules:

1. It can be an empty string `""`, or any single character that is not `"("` or `")"`.
2. A string can be considered a VPS if it is the concatenation of two VPS's, denoted as `AB`.
3. A string is also a VPS if it is of the form `(A)`, where `A` is itself a VPS.

Beyond this, the problem introduces the concept of **nesting depth**. Nesting depth is the maximum level of nested parentheses within any VPS. It's defined as follows:

- `depth("") = 0`, for an empty string.
- `depth(C) = 0`, where `C` is any single character other than parentheses.
- `depth(A + B)` is the maximum depth between two VPS's `A` and `B`.
- `depth("(" + A + ")")` is the depth of `A` plus one, accounting for the additional level of nesting created by surrounding `A` with parentheses.

The task is to calculate and return the nesting depth of a given string `s`, which is guaranteed to be a VPS.

Intuition

The solution to determining the nesting depth of a string involves keeping track of the current level of parenthetical depth while iterating through the string character by character. Whenever an opening parenthesis `"("` is encountered, we increase the depth, and when a closing parenthesis `")"` is encountered, we decrease the depth.

The intuition behind the solution is as follows:

- A depth tracker variable `d` is initialized to zero.
- An answer variable `ans` is also initialized to zero to keep track of the maximum depth encountered.
- As we iterate over each character `c` of the string `s`:
  - If `c` is an opening parenthesis `(`, we increase the `d` count to indicate that we are going a level deeper.
  - We update the `ans` variable with the maximum between the current `ans` and the updated depth `d`.
  - If `c` is a closing parenthesis `)`, we decrease the `d` count to indicate that we are coming out of a depth level.
- The `ans` will contain the maximum depth achieved throughout the iteration and is returned as the result.

This approach is efficient because it operates in linear time, making a single pass through the string, and requires only a constant amount of extra space.

Solution Approach

The Reference Solution Approach makes use of a simple yet effective algorithm to find the nesting depth of a valid parentheses string. This approach does not require complex data structures or patterns but relies on basic variables to track the progress. The implementation steps are as follows:

1. Initialize two integer variables: `ans` and `d` to zero. `ans` will hold our final result, the maximum depth of nesting, and `d` will keep track of the current depth while iterating through the string.
2. Iterate over each character `c` in the input string `s`.
   - When `c` is an opening parenthesis `(`: a. Increase the depth `d` by 1 because we've entered a new layer of nesting. b. Update the answer `ans` with the greater of `ans` or `d`. This step ensures that `ans` always contains the maximum depth observed so far.
   - When `c` is a closing parenthesis `)`: a. Decrease the depth `d` by 1 because we've exited a layer of nesting.

Here's a more detailed breakdown of the algorithm using the provided Python code:

- `def maxDepth(self, s: str) -> int:` defines a method `maxDepth` that takes a string `s` as an input and returns an integer representing the maximum depth.
- `ans = d = 0` sets up our variables: `ans` to track the maximum depth and `d` to track the current depth.
- `for c in s:` starts a loop over each character in the string.
- `if c == '(':` checks if the current character is an opening parenthesis.
- `d += 1` increments the current depth because an opening parenthesis indicates deeper nesting.
- `ans = max(ans, d)` updates the maximum depth found so far.
- `elif c == ')':` checks if the current character is a closing parenthesis.
- `d -= 1` decrements the current depth because a closing parenthesis indicates that we are stepping out of a level of nesting.
- `return ans` returns the highest depth of nesting that was recorded during the iteration through the string.

By maintaining a count of the current depth level with each parenthesis encountered and recording the maximum depth along the way, the algorithm effectively computes the nesting depth of the valid parentheses string with a time complexity of O(n) and a space complexity of O(1), where n is the length of the string `s`.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the string `s = "(ab((cd)e))"`. Our goal is to determine the maximum nesting depth of this valid parentheses string.

1. We start with `ans = 0` and `d = 0`. These will keep track of the maximum depth and the current depth, respectively.
2. Begin iterating over each character in `s`.
3. First character: `(`:
   - Current depth `d` is increased: `d = 1`.
   - Update `ans`: `ans = max(0, 1) = 1`.
4. Next characters: `ab`, there are no parentheses, so `d` and `ans` remain unchanged.
5. Next character: `(`:
   - Current depth `d` is increased: `d = 2`.
   - Update `ans`: `ans = max(1, 2) = 2`.
6. Next characters: `cd`, there are no parentheses, so `d` and `ans` remain unchanged.
7. Next character: `)`:
   - Current depth `d` is decreased: `d = 1`.
8. Next character: `e`, still no parentheses, so `d` and `ans` remain unchanged.
9. Next character: `)`:
   - Current depth `d` is decreased: `d = 0`.
   - The string ends here, and `ans` holds the maximum depth encountered, which is `2`.

At the end of this process, the maximum depth `ans` is `2`, which is the correct nesting depth of the example string `s`. Throughout this iterative process, we have updated `ans` whenever a greater depth was achieved and correctly maintained the current depth by incrementing and decrementing `d` at the opening and closing parentheses, respectively. The linear scan allowed us to calculate the nesting depth in a single pass, fulfilling the efficient performance as outlined in the solution approach.

Solution Implementation

Python

```python
class Solution:
    def maxDepth(self, s: str) -> int:
        # Initialize variables to store the current depth and maximum depth
        max_depth = current_depth = 0

        # Iterate through each character in the string
        for char in s:
            if char == '(':  # If the character is an opening bracket
                current_depth += 1  # Increase the current depth by 1
                max_depth = max(max_depth, current_depth)  # Update the max depth if needed
            elif char == ')':  # If the character is a closing bracket
                current_depth -= 1  # Decrease the current depth by 1

        # Return the maximum depth encountered
        return max_depth
```

Java

```java
class Solution {
    public int maxDepth(String s) {
        int maxDepth = 0; // This will store the maximum depth of the parentheses
        int currentDepth = 0; // This will keep track of the current depth

        // Iterating over each character in the string
        for (int i = 0; i < s.length(); ++i) {
            char c = s.charAt(i); // Get the current character from the string

            if (c == '(') {
                // If the character is an opening parenthesis, we increase the current depth
                currentDepth++;

                // Update the maximum depth if the current depth is greater than the maxDepth observed so far
                maxDepth = Math.max(maxDepth, currentDepth);
            } else if (c == ')') {
                // If the character is a closing parenthesis, we decrease the current depth
                currentDepth--;
            }
            // We ignore all other characters
        }

        // Returning the maximum depth of nesting of the parentheses encountered in the string
        return maxDepth;
    }
}
```

C++

```cpp
#include <algorithm> // For using the max function
#include <string>    // For using the string type

class Solution {
public:
    // Function to find the maximum depth of parentheses in a given string `s`.
    int maxDepth(string s) {
        int maxDepth = 0; // To store the maximum depth encountered
        int currentDepth = 0; // To track the current depth of open parentheses

        // Iterate through each character in the string
        for (char& c : s) {
            if (c == '(') {
                // Increment the current depth when an opening parenthesis is encountered
                currentDepth++;
                // Update the maximum depth encountered so far
                maxDepth = std::max(maxDepth, currentDepth);
            } else if (c == ')') {
                // Decrement the current depth when a closing parenthesis is encountered
                currentDepth--;
            }
            // No action on other characters
        }
        // Return the maximum depth of opened parentheses
        return maxDepth;
    }
};
```

TypeScript

```typescript
// Function to find the maximum nesting depth of parentheses in a given string.
function maxDepth(s: string): number {
    let maxDepth = 0; // This will keep track of the maximum depth encountered
    let currentDepth = 0; // This will keep track of the current depth

    // Iterate over each character in the input string
    for (const char of s) {
        // If the character is an opening parenthesis, increase the current depth
        if (char === '(') {
            currentDepth++;
            // Update the maximum depth if the current depth exceeds it
            maxDepth = Math.max(maxDepth, currentDepth);
        } else if (char === ')') {
            // If the character is a closing parenthesis, decrease the current depth
            currentDepth--;
        }
        // We ignore all other characters
    }

    return maxDepth; // Return the maximum depth encountered
}
```


```python
class Solution:
    def maxDepth(self, s: str) -> int:
        # Initialize variables to store the current depth and maximum depth
        max_depth = current_depth = 0

        # Iterate through each character in the string
        for char in s:
            if char == '(':  # If the character is an opening bracket
                current_depth += 1  # Increase the current depth by 1
                max_depth = max(max_depth, current_depth)  # Update the max depth if needed
            elif char == ')':  # If the character is a closing bracket
                current_depth -= 1  # Decrease the current depth by 1

        # Return the maximum depth encountered
        return max_depth
```


Time and Space Complexity

Time Complexity

The time complexity of the code is a function of the number of characters in the input string, `s`. The code consists of a single `for` loop that goes through each character of `s` exactly once, performing `O(1)` operations for each character - increasing or decreasing the depth `d` and updating the maximum depth `ans`. Therefore, the time complexity is `O(n)`, where `n` is the length of the input string `s`.

Space Complexity

The space complexity of the code is `O(1)`. This is because the code uses a fixed number of integer variables `ans` and `d`, regardless of the input size. No additional structures that grow with input size are used, which means the space used by the algorithm does not depend on the input size.