32. Longest Valid Parentheses

Hard Stack String Dynamic Programming

Problem Description

In this problem, we are given a string that consists only of the characters '(' and ')'. The goal is to find the length of the longest substring that represents a valid sequence of parentheses. A valid parentheses string is one where every '(' has a matching ')' and the pairs are well-nested. For example, in the string "(()", only the substring "()" is valid and has a length of 2.

Intuition

The key to solving this problem lies in the fact that a valid parentheses string must end with a ')' character. To systematically approach this, we use dynamic programming (DP). The solution builds up information about the substrings ending at each position in the input string, using an array (or list) to record the results.

The intuition behind the dynamic programming solution is this: We define a DP array where the ith entry, `f[i]`, represents the length of the longest valid parentheses substring ending at the index `i - 1` in the input string. To fill in this array, we consider two main cases:

1. If the character at position `i - 1` is '(', then no valid substring can end with this character, so `f[i]` would be 0.

2. If the character at position `i - 1` is ')', this is where it gets interesting. We look at two possible scenarios:

   a. If the character right before it (at `i - 2`) is '(', we have a pair "()", so the length of the longest valid substring ending at `i - 1` would be `f[i - 2] + 2`, since we're adding these two characters to whatever we had before.

   b. If the character at `i - 2` is also ')', the situation is a bit more complex. This might be a continuation of a previously started valid string. We first check that there's a valid string ending just before the current one (at `i - f[i - 1] - 2`). If there is, and the character at the beginning of this string (at `i - f[i - 1] - 2`) is '(', then the current ')' can close this, forming a new valid string, so we append the lengths: `f[i - 1] + 2 + f[i - f[i - 1] - 2]`.

With this DP array filled out, all we need to do is to find the maximum value in it, which represents the length of the longest valid parentheses substring in the entire input string. The code iterates through the string only once and fills the DP array as it goes, making it an efficient solution.

Solution Approach

The solution presented here is a prime example of dynamic programming. Specifically, it showcases a bottom-up iterative approach to solve the problem of finding the longest valid parentheses substring. Here's how the approach is implemented:

Firstly, we initialize a list `f` with a length of `n + 1`, where `n` is the length of the input string `s`, and fill it with zeros. This list will hold our dynamic programming table, where `f[i]` represents the length of the longest valid parentheses substring that ends at index `i - 1`.

As we iterate over the string (1-indexed for easy reference), we only need to consider elements in `s` that are closing parentheses `')'`. This is because valid substrings can only end with a closing parenthesis.

For each closing parenthesis at index `i`, we have two main cases:

- **Case 1**: The previous character is an opening parenthesis `'('`. This means we've found a simple pair of parentheses "`()`". The length of the longest valid substring ending at `i` is `f[i - 2] + 2`. This consists of the length of whatever valid substring ends right before this pair plus the length of this pair (2).

- **Case 2**: The previous character is also a closing parenthesis `')'`. Here, we have to look back and check if there's a matching opening parenthesis that can pair with our current one. To find it, we subtract the length of the valid substring ending just before (at `f[i - 1]`) from `i` and go back one more step (hence `i - f[i - 1] - 2`). If we find that there's an opening parenthesis at that position, we can extend the length of the valid substring ending there by adding the length of the valid substring ending at our current position plus 2 for the current pair "()", and also add the value at `f[j - 1]`, where `j` is the index we just found to check availability of a valid opening parenthesis. This step accounts for concatenating a new valid pair to an existing valid substring.

By repeatedly applying the above logic as we scan through the string, we build up our DP table. Once we have processed the entire string, we can simply return `max(f)`, which will give us the length of the longest valid parentheses substring found.

The approach effectively breaks down the problem into manageable subproblems. It ensures that every position in the string is only calculated once, giving us an overall time complexity of `O(n)` where `n` is the length of the string. The space complexity is also `O(n)` due to the additional list used for storing the lengths of longest valid substrings.

Example Walkthrough

Let's apply the solution approach to a small example and illustrate how it works.

Suppose we are given the string `s = "(()())"`.

We first initialize a list `f` with a length of `n + 1 = 7` (since the length of `s` is 6), and fill it with zeros: `f = [0, 0, 0, 0, 0, 0, 0]`.

Now, we will iterate over the string, starting from index 1:

1. `i = 1`: `s[i - 1] = '('`, so we do nothing because a valid substring cannot end with '('. `f = [0, 0, 0, 0, 0, 0, 0]`

2. `i = 2`: `s[i - 1] = '('`, we do nothing again for the same reason. `f = [0, 0, 0, 0, 0, 0, 0]`

3. `i = 3`: `s[i - 1] = ')'`, previous char is `'('`, so we have "()", and `f[i] = f[1] + 2 = 2`. `f = [0, 0, 2, 0, 0, 0, 0]`

4. `i = 4`: `s[i - 1] = '('`, do nothing. `f = [0, 0, 2, 0, 0, 0, 0]`

5. `i = 5`: `s[i - 1] = ')'`, the previous character is `'('`, so "()" again, `f[i] = f[3] + 2 = 4` (because we are adding the pair `()` to what we had at `f[3]`). `f = [0, 0, 2, 0, 4, 0, 0]`

6. `i = 6`: `s[i - 1] = ')'`, the previous character is `')'`, we check `i - f[i - 1] - 2 = 6 - 4 - 2 = 0`, at position `0` we have `'('`, so we have a longer valid substring that spans the whole current string, `f[i] = f[4] + 2 + f[6 - 4 - 2] = 4 + 2 + 0`. `f = [0, 0, 2, 0, 4, 0, 6]`

Now our DP list `f` tells us that the longest valid substring ending at any position in `s`. The max value in `f` is `6`, which is the length of the longest valid parentheses substring in `s`.

This walkthrough demonstrates the bottom-up DP technique used to find the longest valid parentheses substring in linear time complexity, which is `O(n)`.

Python Solution

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        # Get the length of the input string
        string_length = len(s)
        # Initialize a DP array with zeros, one extra for the base case
        dp = [0] * (string_length + 1)

        # Loop over the characters of string starting from index 1 for convenience
        for index, char in enumerate(s, 1):
            # We look for closing parentheses, as they mark possible ends of valid substrings
            if char == ")":
                # If the previous char is '(', it's a pair "()"
                if index > 1 and s[index - 2] == "(":
                    # Add 2 to the result two positions ago in dp array
                    dp[index] = dp[index - 2] + 2
                else:
                    # Get the index of the potential matching '('
                    match_index = index - dp[index - 1] - 1
                    # Make sure match_index is within bounds and check for '('
                    if match_index > 0 and s[match_index - 1] == "(":
                        # Add the length of the valid substring ending right before the current one,
                        # plus two for the '()' just found, plus length of valid substring before the pair
                        dp[index] = dp[index - 1] + 2 + dp[match_index - 1]

        # Return the maximum length of valid parentheses found
        return max(dp)

```

Java Solution

```java
class Solution {
    public int longestValidParentheses(String s) {
        int strLength = s.length();
        int[] validLengths = new int[strLength + 1]; // Array to store the length of valid parentheses substrings at each index.
        int maxValidLength = 0; // The maximum length of valid parentheses found.

        // Iterate starting from the second character as we need to check pairs.
        for (int i = 2; i <= strLength; ++i) {
            // Check if the current character is a closing parenthesis.
            if (s.charAt(i - 1) == ')') {
                // If the previous character is an opening parenthesis, it forms a valid pair.
                if (s.charAt(i - 2) == '(') {
                    validLengths[i] = validLengths[i - 2] + 2; // Add 2 for the valid '()' pair.
                } else {
                    // If the previous character is also a closing parenthesis,
                    // find the position before the valid substring that starts
                    // just before the current one.
                    int prevIndex = i - validLengths[i - 1] - 1;
                    // Check if there's a matching opening parenthesis.
                    if (prevIndex > 0 && s.charAt(prevIndex - 1) == '(') {
                        validLengths[i] = validLengths[i - 1] + 2 + validLengths[prevIndex - 1]; // Add lengths of valid substrings
                    }
                }
                // Update the maximum length if needed.
                maxValidLength = Math.max(maxValidLength, validLengths[i]);
            }
        }

        return maxValidLength; // Return the maximum length of valid parentheses.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <cstring>

class Solution {
public:
    int longestValidParentheses(string s) {
        int length = s.size();

        // Create a dynamic programming vector with 0 initialization
        vector<int> dp(length + 1, 0);

        // Iterate over the string, starting at the second character
        for (int i = 2; i <= length; ++i) {
            // If the current character is a closing parenthesis
            if (s[i - 1] == ')') {
                // If the previous character is an opening parenthesis
                if (s[i - 2] == '(') {
                    // Add 2 to the dynamic programming array at position i
                    // This forms a pair with the previous '('
                    dp[i] = dp[i - 2] + 2;
                } else {
                    // Calculate the index before the previous valid substring
                    int previousIndex = i - dp[i - 1] - 1;
                    // If there's an opening parenthesis before the previous valid substring
                    if (previousIndex > 0 && s[previousIndex - 1] == '(') {
                        // Add the lengths of the currently found valid substring, the one before it,
                        // and the one before the opening parenthesis
                        dp[i] = dp[i - 1] + 2 + dp[previousIndex - 1];
                    }
                }
            }
        }

        // Return the maximum length found in the dp array
        return *max_element(dp.begin(), dp.end());
    }
};

```

Typescript Solution

```typescript
// This function calculates the length of the longest valid (well-formed) parentheses substring.
function longestValidParentheses(s: string): number {
    const lengthOfString: number = s.length;
    // Initialize an array to record the length of the longest valid parentheses ending at each position.
    const dp: number[] = new Array(lengthOfString + 1).fill(0);
    let maxValidLength: number = 0; // Keep track of the maximum valid length.

    // Iterate through the string starting from the second character position.
    for (let i = 2; i <= lengthOfString; i++) {
        // Check if the current character is a closing parenthesis.
        if (s[i - 1] === ')') {
            // If previous character is an opening parenthesis, it's a valid pair.
            if (s[i - 2] === '(') {
                dp[i] = dp[i - 2] + 2;
            } else {
                // Check if the substring before the last valid parentheses is an opening parenthesis.
                const previousIndex: number = i - dp[i - 1] - 1;
                if (previousIndex > 0 && s[previousIndex - 1] === '(') {
                    dp[i] = dp[i - 1] + 2 + dp[previousIndex - 1];
                }
            }
        }
        // Update the maximum valid length found so far.
        maxValidLength = Math.max(maxValidLength, dp[i]);
    }

    // Return the maximum valid length of the parentheses found.
    return maxValidLength;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the string `s`. This is because the algorithm iterates through all characters of the input string once, and for each character, it performs a constant amount of work.

The space complexity of the code is `O(n)` as well, due to the auxiliary space used by the list `f`, which has a length of `n + 1`. Each element of the list `f` stores the length of the longest valid substring ending at that position, thereby requiring linear space relative to the input string length.