# 1021. Remove Outermost Parentheses

Easy Stack String

## Problem Description

The problem requires us to remove the outermost parentheses from each primitive valid parentheses string within a given valid parentheses string `s`. A primitive valid parentheses string is one that cannot be decomposed into two nonempty valid parentheses strings. For example, in the string `"((()())(()))"`, there are two primitive strings: `"(()())"` and `"(())"`. After removing the outermost parentheses from these substrings, they become `"()()"` and `"()"` respectively, so the final result is `"()()()"`.

## Intuition

The key to solving this problem lies in finding a way to track the boundaries of the primitive strings within the input string `s`. Since a valid parentheses string is balanced, we can increment a counter for each opening parenthesis and decrement it for each closing parenthesis.

Here is how we arrive at the solution:

- We define a counter `cnt` to track the level of nesting of the parentheses.
- We iterate over the characters in the input string.
- For each character:
  - If it's an opening parenthesis `'('`:
    - We increment the counter `cnt`.
    - If `cnt` is more than 1, it means this `'('` is not the outermost parenthesis, so we include it in the result.
  - If it's a closing parenthesis `')'`:
    - We decrement the counter `cnt` first, because we want to check if it **was** greater than 1 before this closing parenthesis.
    - If `cnt` is still greater than 0 after decrementing, it's not the outermost parenthesis, so we include it in the result.
- We join all the parentheses we've added to the result list to get the final string without outermost parentheses.

By maintaining the nesting level, we ensure we only remove the outermost parentheses and keep the structure of the remaining inner valid parentheses intact.

## Solution Approach

The solution implements a simple algorithm that relies on a counter to track the level of nesting of the parentheses. Let's walk through the implementation:

```
class Solution:
    def removeOuterParentheses(self, s: str) -> str:
        ans = []  # This list will store the characters to build the final answer.
        cnt = 0   # This counter tracks the depth of the parentheses.

        for c in s:    # Iterate over all the characters in the input string.
            if c == '(':  # If the character is an opening parenthesis...
                cnt += 1  # Increase the counter.
                if cnt > 1:  # If the counter is greater than 1...
                    ans.append(c)  # ...it's not an outer parenthesis, so add it to the answer.
            else:  # If the character is a closing parenthesis...
                cnt -= 1  # Decrease the counter first.
                if cnt > 0:  # If the counter is still greater than 0...
                    ans.append(c)  # ...it's not an outer parenthesis, so add it to the answer.

        return ''.join(ans) # Join all characters to get the final string.
```

The algorithm uses a list `ans` as an auxiliary data structure for building the resulting string, which avoids the higher cost of string concatenation on each iteration.

In terms of space complexity, the solution has a space complexity of $O(n)$, where $n$ is the length of the input string, since in the worst case, we might store almost the entire string in the `ans` list (minus the outer parentheses).

In terms of time complexity, the solution runs in $O(n)$ time. We iterate through each character in the string exactly once, with constant time operations for each character.

The pattern used here falls under the stack pattern often seen in problems involving string manipulation or parentheses validation. However, instead of using a stack, the solution cleverly employs a counter to simulate the stack's behaviour, as we are only interested in the level of nesting and not the actual sequence of parentheses that led to that nesting level.

### Example Walkthrough

Let's walk through a small example using the provided solution approach. We'll take the input string `"(()())(())"` as our example.

- Initialize the result list `ans = []` and a counter variable `cnt = 0`.
- Start iterating over the string:
  - First character is `'('`: `cnt` becomes `1`. Since `cnt` is not more than `1`, we do not add to `ans`.
  - Second character `'('`: `cnt` becomes `2`. Now `cnt > 1`, so we add `'('` to `ans` → `ans = ['(']`.
  - Third character `')'`: `cnt` decreases to `1` but we check it before decreasing, so it was `2`, add `')'` → `ans = ['(', ')']`.
  - Fourth character `'('`: `cnt` increases to `2`, added to `ans` → `ans = ['(', ')', '(']`.
  - Fifth character `')'`: Decrease `cnt` to `1`, and since it was `2`, add `')'` → `ans = ['(', ')', '(', ')']`.
  - Now we reach the end of the first primitive string, and the sixth character is `'('`: This is the start of the next primitive string. `cnt` becomes `1` again, no addition to `ans`.
  - Seventh character `'('`: `cnt` increases to `2`, which means it's not outer, add `'('` → `ans = ['(', ')', '(', ')', '(']`.
  - Eighth character `')'`: Decrease `cnt` to `1`, add `')'` → `ans = ['(', ')', '(', ')', '(', ')']`.
- Continue until the end following the same logic. As we finish iterating, we have `ans = ['(', ')', '(', ')', '(', ')']`.
- Join the characters in `ans` to form the final answer. `return ''.join(ans)` would result in the string `"()()()"`.

This example demonstrates the efficiency of the approach at handling each character of the input string and effectively maintaining the balance of the parentheses without the need for a stack. The counter `cnt` acts as a simplistic stack that keeps track of nesting levels, which helps decide whether a parenthesis is outermost or not. The final string `"()()()"` is the input string without the outermost parentheses.

## Solution Implementation

### Python

```
class Solution:
    def removeOuterParentheses(self, input_string: str) -> str:
        # Initialize an empty list to store the characters of the resulting string
        result = []

        # Initialize a count variable to keep track of the balance between open and close parentheses
        balance_count = 0

        # Iterate through each character in the input string
        for char in input_string:
            # If the current character is an open parenthesis
            if char == '(':
                # Increase the balance count
                balance_count += 1

                # Add the open parenthesis to the result if it's not part of an outer pair
                if balance_count > 1:
                    result.append(char)
            # If the current character is a close parenthesis
            else:
                # Decrease the balance count
                balance_count -= 1

                # Add the close parenthesis to the result if it's not part of an outer pair
                if balance_count > 0:
                    result.append(char)

        # Join the characters in the result list to form the final string without outer parentheses
        return ''.join(result)
```

### Java

```
class Solution {

    // Method to remove the outermost parentheses from every primitive string in the input string
    public String removeOuterParentheses(String s) {
        // StringBuilder to build the result string
        StringBuilder result = new StringBuilder();

        // Counter to keep track of the balance of parentheses
        int parenthesesCount = 0;

        // Iterate through each character of the input string
        for (int i = 0; i < s.length(); i++) {
            char currentChar = s.charAt(i); // Current character being processed

            // If the current character is an opening parenthesis ('(')
            if (currentChar == '(') {
                // Increment the count. If it's not the first '(' (means it's not outer), append to result
                if (parenthesesCount > 0) {
                    result.append(currentChar);
                }
                parenthesesCount++;
            }
            // If the current character is a closing parenthesis (')')
            else {
                // Decrement the count before the check to see if it's a non-outer ')'
                parenthesesCount--;

                // If the count doesn't represent the outer parenthesis, append to result
                if (parenthesesCount > 0) {
                    result.append(currentChar);
                }
            }
            // No need to handle the else case, since valid inputs only contain '(' and ')'
        }

        // Return the built string which contains no outer parentheses
        return result.toString();
    }
}
```

### C++

```
class Solution {
public:
    // Method to remove the outermost parentheses in each set of valid parentheses.
    string removeOuterParentheses(string s) {
        string result; // This will store the final answer.
        int depth = 0; // This keeps track of the depth of the parentheses.

        // Iterate over each character in the input string.
        for (char& c : s) {
            if (c == '(') {
                // If the character is an opening parenthesis, increase the depth.
                ++depth;
                // Only add the opening parenthesis to the result if the depth is more than 1,
                // because the first opening parenthesis is part of the outermost parentheses.
                if (depth > 1) {
                    result.push_back(c);
                }
            } else {
                // If the character is a closing parenthesis, decrease the depth first.
                --depth;
                // Only add the closing parenthesis to the result if the depth is not zero after decrementing,
                // because the last closing parenthesis corresponds to the opening one that wasn't added.
                if (depth > 0) {
                    result.push_back(c);
                }
            }
        }
        return result; // Return the final string with outermost parentheses removed.
    }
};
```

### TypeScript

```
// This function removes the outermost parentheses from each primitive string within the given string
function removeOuterParentheses(s: string): string {
    let result = ''; // Initialize an empty string to hold the result
    let depth = 0;   // This variable keeps track of the depth of the parentheses

    // Loop through each character in the input string
    for (const char of s) {
        if (char === '(') {
            depth++;  // Increment depth for an opening parenthesis
            if (depth !== 1) {
                // If the depth is not 1, it is not an outer parenthesis, so append it to the result
                result += char;
            }
        } else if (char === ')') {
            depth--;  // Decrement depth for a closing parenthesis
            if (depth !== 0) {
                // If the depth is not 0 after decrementing, it is not an outer parenthesis, so append it to the result
                result += char;
            }
        }
    }

    return result; // Return the modified string without the outermost parentheses
}


class Solution:
    def removeOuterParentheses(self, input_string: str) -> str:
        # Initialize an empty list to store the characters of the resulting string
        result = []

        # Initialize a count variable to keep track of the balance between open and close parentheses
        balance_count = 0

        # Iterate through each character in the input string
        for char in input_string:
            # If the current character is an open parenthesis
            if char == '(':
                # Increase the balance count
                balance_count += 1

                # Add the open parenthesis to the result if it's not part of an outer pair
                if balance_count > 1:
                    result.append(char)
            # If the current character is a close parenthesis
            else:
                # Decrease the balance count
                balance_count -= 1

                # Add the close parenthesis to the result if it's not part of an outer pair
                if balance_count > 0:
                    result.append(char)

        # Join the characters in the result list to form the final string without outer parentheses
        return ''.join(result)
```

## Time and Space Complexity

The time complexity of the provided code can be analyzed by looking at the number of operations it performs relative to the input string length $n$.

- The for-loop iterates over each character of the string `s` exactly once, giving us $n$ iterations.
- Within the for-loop, the operations consist of simple arithmetic (incrementing or decrementing the `cnt` variable) and basic conditional checks, which all run in constant time, $O(1)$.
- The append operation to the `ans` list takes $O(1)$ time for each operation due to the dynamic array implementation of Python lists.

Putting it all together, the code's main contributing factor for time complexity is the single pass through the string, giving us a time complexity of $O(n)$.

Space complexity is determined by the extra space used by the program which is not part of the input space.

- The `cnt` variable uses constant space $O(1)$.
- The `ans` list is the main auxiliary space usage, which in the worst case will store all the characters of the string except the outermost parentheses. The `ans` list thus grows with the size of the input and can be considered $O(n)$ in the worst-case scenario.

Therefore, the space complexity of the code is $O(n)$ where $n$ is the length of the input string `s`.