# 1111. Maximum Nesting Depth of Two Valid Parentheses Strings

Medium Stack String Leetcode Link

## Problem Description

The problem requires us to operate on a string that is said to be a valid parentheses string (VPS). A VPS is defined by the following properties:

- It consists of "(" and ")" characters only.
- It can be an empty string.
- It can be the concatenation of two valid parentheses strings, `A` and `B`.
- It can be a string that encapsulates a valid parentheses string within parentheses, like "(A)".

The concept of nesting depth `depth(S)` is also introduced, which is calculated as follows:

- `depth("")` (the depth of an empty string) is 0.
- `depth(A + B)` (the depth of the concatenation of two VPS's) is the maximum of the depths of `A` and `B`.
- `depth("(" + A + ")")` (the depth when a VPS `A` is within parentheses) is `1 + depth(A)`.

The task is to split a given VPS `seq` into two disjoint subsequences `A` and `B`. Both `A` and `B` should themselves be VPS's, and the total length of `A` and `B` should be equal to the length of `seq`. Out of all possible `A` and `B`, we need to find the pair that minimizes the maximum of their depths.

The output should be an array `answer` where each element is either 0 or 1. `answer[i]` should be 0 if the character `seq[i]` is part of the subsequence `A`, otherwise it's part of `B`. Though there can be multiple correct `A` and `B` pairs, any valid solution is acceptable.

## Intuition

The intuition behind the solution is related to the observation that in a valid parentheses string, a subsequence `A` can be formed by taking some of the "left parentheses" `(` and the matching "right parentheses" `)` to keep `A` as a VPS, while the rest can form the subsequence `B`. By doing so, we can potentially balance the depth between `A` and `B`.

A straightforward approach to split the pairs between `A` and `B` is to alternate assignments of the parentheses as we traverse the given `seq`. Each time we encounter a "left parentheses" `(`, we assign it to `A` if we are at an "even level" and to `B` if we are at an "odd level". When we encounter a "right parentheses" `)`, we step back one level and then perform the similar alternating assignment.

This alternating assignment helps ensure that we don't create too deep a nesting in any of the subsequences, hence minimizing the maximum depth between them.

To implement this, we keep a counter `x` to track the current level of depth during the traversal, and toggle the assignment of parentheses based on the current level's parity (even or odd) indicated by `x & 1`. An even `x` represents that we assign to sequence `A` (`answer[i]` is 0), and an odd `x` represents that we assign to sequence `B` (`answer[i]` is 1).

## Solution Approach

The solution uses a simple algorithm that iterates through the given VPS string `seq` and assigns each parenthesis to either subsequence `A` or `B` based on the current depth level. Here's how it's implemented:

- Initialize an array `ans` with the same length as `seq` to store the assignment of each parenthesis to subsequence `A` (represented by 0) or `B` (represented by 1).
- Initialize a variable `x` that will keep track of the current depth level as we walk through the string. This variable will increment when a "left parentheses" `(` is encountered and decrement when a "right parentheses" `)` is encountered.
- Iterate over the characters of `seq` using `enumerate` to have both index `i` and character `c`. For each character:
  - If `c` is a "left parentheses" `(`, determine if the current depth level `x` is even or odd by checking `x & 1`. If it's even (`x & 1` is 0), it means we are at an even depth level, so `ans[i]` is set to 0, indicating the parenthesis belongs to `A`. If it's odd (`x & 1` is 1), `ans[i]` is set to 1, meaning it belongs to `B`. After assigning the parenthesis, increment the depth level `x`.
  - If `c` is a "right parentheses" `)`, first decrement the depth level `x` because we are closing a parenthesis. Then check `x & 1` and assign `ans[i]` in the same way as if it was a "left parentheses".
- Once the iteration is complete, return the `ans` array which now represents the VPS split between `A` and `B` in such a way that the maximum depth of both is minimized.

This algorithmic approach ensures that the split of parentheses between `A` and `B` keeps their respective depths balanced, thus minimizing the maximum depth of any VPS that could be formed by them. No additional data structures are needed beyond the `ans` array for the output and a single integer variable for depth tracking.

### Example Walkthrough

Let's consider the VPS sequence `seq = "(()())"`

We wish to split this sequence into two disjoint subsequences `A` and `B` such that the maximum depth of `A` and `B` is minimized.

The algorithm goes as follows:

1. Initialize `ans` to be an array of the same length as `seq`. In this case, `ans = [0, 0, 0, 0, 0, 0]`.
2. Initialize our depth counter, `x`, to `0`.
3. Now, we iterate through the `seq` string character by character.

Here's a step-by-step walkthrough:

- `i = 0`, `c = '('`. Since `x` is `0` (even), `ans[i]` is set to `0`. `x` increments to `1`. Now, `ans = [0, 0, 0, 0, 0, 0]`.
- `i = 1`, `c = '('`. `x` is `1` (odd), `ans[i]` is set to `1`. `x` increments to `2`. Now, `ans = [0, 1, 0, 0, 0, 0]`.
- `i = 2`, `c = ')'`. Before assignment, we decrement `x` to `1`. Since `x` is now `1` (odd), `ans[i]` is set to `1`. Now, `ans = [0, 1, 1, 0, 0, 0]`.
- `i = 3`, `c = '('`. `x` is `1` (odd), `ans[i]` is set to `1`. `x` increments to `2`. Now, `ans = [0, 1, 1, 0, 0, 0]`.
- `i = 4`, `c = ')'`. We decrement `x` to `1` before assigning. `x` is `1` (odd), `ans[i]` is set to `1`. Now, `ans = [0, 1, 1, 0, 1, 0]`.
- `i = 5`, `c = ')'`. We decrement `x` to `0` before assigning. Since `x` is now `0` (even), `ans[i]` is set to `0`. Now, `ans = [0, 1, 1, 0, 1, 0]`.

So the final `ans` representing the split is `[0, 1, 1, 0, 1, 0]`. Here, `A` consists of the parentheses at positions `0, 3, and 5` corresponding to the VPS "(()", and `B` consists of the parentheses at positions `1, 2, and 4` corresponding to the VPS "()()". Both `A` and `B` are valid parentheses strings and the overall maximum depth is minimized.

## Python Solution

```python
from typing import List

class Solution:
    def maxDepthAfterSplit(self, seq: str) -> List[int]:
        # Initialize an array to store the assignment of depths to each parenthesis.
        assigned_depth = [0] * len(seq)

        # Declare a variable to keep track of the current depth level.
        depth_level = 0

        # Iterate over each character in the sequence alongside its index.
        for index, char in enumerate(seq):
            if char == "(":
                # If the parenthesis is an opening one, determine the depth.
                # We use bitwise AND with 1 to alternate between 0 and 1.
                assigned_depth[index] = depth_level & 1
                # Increment depth level since we've encountered an opening parenthesis.
                depth_level += 1
            else:
                # If the parenthesis is a closing one, first decrement the depth level.
                depth_level -= 1
                # After decreasing the depth level, determine the depth for the closing parenthesis.
                assigned_depth[index] = depth_level & 1

        # Return the final array with the assigned depths.
        return assigned_depth

```

## Java Solution

```java
class Solution {
    // This method splits the maximum depth of balanced sub-sequences from the given sequence
    public int[] maxDepthAfterSplit(String seq) {
        // Get the length of the sequence
        int n = seq.length();
        // Initialize the answer array to hold the group assignment of each character in 'seq'
        int[] answer = new int[n];
        // Loop through the characters of 'seq'
        for (int i = 0, depthLevel = 0; i < n; ++i) {
            // If the current character is an opening parenthesis '('
            if (seq.charAt(i) == '(') {
                // Assign the current group based on the parity of 'depthLevel' and increment 'depthLevel'
                answer[i] = depthLevel++ & 1;
            } else {
                // If it's a closing parenthesis ')', decrement 'depthLevel' first
                // then assign the current group based on the parity of 'depthLevel'
                answer[i] = --depthLevel & 1;
            }
        }
        // Return the array containing group assignments for each character
        return answer;
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    // Function to assign depths after split based on the sequence of parentheses
    std::vector<int> maxDepthAfterSplit(std::string seq) {
        int size = seq.size(); // Get the size of the sequence
        std::vector<int> answer(size); // Initialize a vector to store the answer with the same size as the input
        for (int i = 0, depth = 0; i < size; ++i) {
            if (seq[i] == '(') {
                // If the character is '(', increment the depth and assign the current depth modulo 2 to the answer.
                // This is because depth alternates for better balancing the depth after splitting.
                answer[i] = depth++ & 1;
            } else {
                // For ')', decrement the depth first because we are ending a level of depth
                // before assigning to answer.
                answer[i] = --depth & 1;
            }
            // "& 1" effectively takes the least significant bit of the depth,
            // which is equivalent to depth % 2 (but faster), to alternate between 0 and 1.
        }
        return answer; // Return the computed vector of depths.
    }
};

```

## Typescript Solution

```typescript
function maxDepthAfterSplit(seq: string): number[] {
    const sequenceLength = seq.length;  // The length of the input string
    const answer: number[] = new Array(sequenceLength);  // Initialize the answer array with the same length as input

    // Initialize variables for iteration and depth calculation
    let depth = 0;  // Used to track the depth of nested parentheses

    // Loop through each character in the input string
    for (let index = 0; index < sequenceLength; ++index) {
        if (seq[index] === '(') {
            // If the current character is an opening parenthesis, determine which group it belongs to
            // Use bitwise AND with 1 to alternate groups (0 and 1) as depth increases
            answer[index] = depth++ & 1;
        } else {
            // If the current character is a closing parenthesis, decrement depth first before determining the group
            // Again, use bitwise AND with 1 to alternate groups
            answer[index] = --depth & 1;
        }
    }

    return answer;  // Return the populated answer array
}

```

## Time and Space Complexity

The given code snippet takes a string `seq`, representing a sequence of parentheses, and returns a list of integers that correspond to the depth of each parenthesis in a way such that the sequence is split into two sequences A and B, with both being valid parentheses strings and depth ideally balanced.

### Time Complexity

The time complexity of this code is `O(n)`. This is because there is a single loop that iterates through the string `seq`, which is of length `n`. Within this loop, all operations (assigning `ans[i]`, incrementing or decrementing `x`, and bitwise comparison with 1) are constant time operations, `O(1)`. Since these constant time operations are repeated `n` times, the overall time complexity remains linear.

### Space Complexity

The space complexity is also `O(n)`. The additional space used by the algorithm is primarily for the output list `ans`, which has the same length as the input string `seq`. There are no other data structures that grow with the size of the input, and variables like `x` and `i` use constant space. Therefore, the space complexity is directly proportional to the input size.