22. Generate Parentheses

Medium String Dynamic Programming Backtracking Leetcode Link

Problem Description

The problem requires us to generate all possible combinations of well-formed parentheses given `n` pairs. A well-formed combination of parentheses means that each opening bracket "(" has a corresponding closing bracket ")", and they are correctly nested. To better understand, for `n=3`, one such correct combination would be `"((()))"`, whereas `"(()"` or `"())("` would be incorrect formations.

Intuition

To arrive at the solution, we need to think about how we can ensure we create well-formed parentheses. For that, we use Depth First Search (DFS), which is a recursive method to explore all possible combinations of the parentheses.

1. We start with an empty string and at each level of the recursion we have two choices: add an opening parenthesis "(" or a closing parenthesis ")".
2. However, we have to maintain the correctness of the parentheses. This means we cannot add a closing parenthesis if there are not enough opening ones that need closing.
3. We keep track of the number of opening and closing parentheses used so far. We are allowed to add an opening parenthesis if we have not used all `n` available opening parentheses.
4. We can add a closing parenthesis if the number of closing parentheses is less than the number of opening parentheses used. This ensures we never have an unmatched closing parenthesis.
5. We continue this process until we have used all `n` pairs of parentheses.
6. When both the opening and closing parentheses counts equal `n`, it means we have a valid combination, so we add it to our list of answers.

The code uses a helper function `dfs` which takes 3 parameters: the number of opening and closing parentheses used so far (`l` and `r`), and the current combination of parentheses (`t`).

By calling this function and starting our recursion with 0 used opening and closing parenthesis and an empty string, we will explore all valid combinations and store them in the list `ans`.

Solution Approach

The solution uses the DFS (Depth First Search) algorithm to generate the combinations. It employs recursion as a mechanism to explore all possible combinations and backtracks when it hits a dead end (an invalid combination).

Here's a step-by-step breakdown of the DFS algorithm as implemented in the provided solution:

1. **Initial Call**: The `generateParenthesis` function initiates the process by calling the nested `dfs` (Depth First Search) function with initial values of zero used opening parentheses (`l`), zero used closing parentheses (`r`), and an empty string for the current combination (`t`).

2. **DFS Function**: This is the recursive function that contains the logic for the depth-first search. It takes three parameters:

   - `l`: The number of opening parentheses used so far.
   - `r`: The number of closing parentheses used so far.
   - `t`: The current combination string formed by adding parentheses.

3. **Base Case**: The recursion has two base cases within the DFS function: a. Invalid Condition: When the number of used opening parentheses `l` is more than `n`, or the closing parentheses `r` is more than `n` or more than `l`, it indicates an incorrect combination. The function returns without doing anything. b. Valid Combination: When both `l` and `r` equal `n`, it indicates that a valid combination of parentheses has been found. The current combination string `t` is added to the solution set `ans`.

4. **Recursive Exploration**: If neither base case is met, the function continues to explore:

   - Adding an opening parenthesis: If not all `n` opening parentheses have been used (`l < n`), the `dfs` function calls itself with `l + 1`, `r`, and appends `"("` to the current string `t`.
   - Adding a closing parenthesis: If the number of closing parentheses used is less than the number of opening parentheses (`r < l`), it implies that there are some unmatched opening parentheses. Thus, the `dfs` function calls itself with `l`, `r + 1`, and appends `")"` to `t`.

By calling these two lines of code, we ensure that we explore the decisions to either add an opening parenthesis or a closing one, thus generating all valid paths.

5. **Storage of Valid Combinations**: The `ans` list is the container that holds all valid combinations. Each time a complete valid combination is generated, it's added to this list. After all recursive calls are completed, `ans` will contain all the possible well-formed parentheses combinations.

6. **Return Result**: Finally, once all possible combinations have been explored, the `ans` list is returned as the result of the `generateParenthesis` function.

This implementation provides a sleek and efficient way to solve the problem of generating all combinations of well-formed parentheses, relying solely on the DFS strategy without needing any additional complex data structures.

Example Walkthrough

Let's consider a small example where `n = 2`, meaning we want to generate all combinations of well-formed parentheses for 2 pairs.

Step 1: Initial Call
The `generateParenthesis` function begins by making an initial call to `dfs` with `l = 0`, `r = 0`, and `t = ""` (an empty string).

Step 2: First Level Recursive Calls
At this stage, we have two choices: add an opening parenthesis or add a closing parenthesis. Since `l < n`, we can add an opening parenthesis. We cannot add a closing parenthesis yet because `r < l` is not satisfied (both `l` and `r` are 0). So, our recursive calls are:

- `dfs(1, 0, "(")`

Step 3: Second Level Recursive Calls
After the first opening parenthesis is added, we are again at a stage where we can choose to add an opening or closing parenthesis. The string is now `"("`.

- For `l < n`, which is true (1 < 2), we add another opening parenthesis: `dfs(2, 0, "((")`.
- We still cannot add a closing parenthesis yet as `r` is not less than `l` (`r < l` is not true).

Step 4: Third Level Recursive Calls
We now have the string `"(("` and `l = 2, r = 0`. We cannot add any more opening parentheses because `l` is not less than `n` anymore (2 is not less than 2). We must add a closing parenthesis now since `r < l` is satisfied. We get:

- `dfs(2, 1, "(()")`

Step 5: Fourth Level Recursive Calls
Our current string is `"(()"` and `l = 2, r = 1`. We still satisfy the condition `r < l`, so we can add another closing parenthesis:

- `dfs(2, 2, "(())")`

Step 6: Base Case Reached
Now `l = 2` and `r = 2`, which equals `n`. We have reached a base case where we have a well-formed combination. This combination `"(())"` is added to our answer set `ans`.

Backtracking
The algorithm will backtrack now and explore other paths, but since `n = 2` and we have used all our opening parentheses, there are no more paths to discover.

Step 7: Return Result
The completed list `ans`, now containing `"(())"`, is returned.

Considering another branch of this example, if we go back to the second level again and instead of adding another opening parenthesis, we decide to add a closing parenthesis:

- After the first level, we have `"("`.
- Add a closing parenthesis: `dfs(1, 1, "()")`, because we can add a closing parenthesis as `r < l`.
- Now, we have `l = 1` and `r = 1`, we can add an opening parenthesis: `dfs(2, 1, "()(")`.
- We can only add a closing parenthesis now: `dfs(2, 2, "()()")`.

So the complete set of combinations for `n = 2` is `"(())"` and `"()()"`.

Python Solution

```python
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        # Helper function for depth-first search
        def backtrack(open_count, close_count, path):
            # If there are more open or more close parens than 'n', or more close parens than open, it's invalid
            if open_count > n or close_count > n or open_count < close_count:
                return
            # When the current path uses all parens correctly, add the combination to the results
            if open_count == n and close_count == n:
                combinations.append(path)
                return
            # Continue the search by adding an open paren if possible
            backtrack(open_count + 1, close_count, path + '(')
            # Continue the search by adding a close paren if possible
            backtrack(open_count, close_count + 1, path + ')')

        # This list will hold all the valid combinations
        combinations = []
        # Start the recursive search with initial counts of open and close parentheses
        backtrack(0, 0, '')
        # Return all the valid combinations found
        return combinations

```

Java Solution

```java
class Solution {
    // List to hold all the valid parentheses combinations
    private List<String> answers = new ArrayList<>();
    // The number of pairs of parentheses
    private int maxPairs;

    /**
     * Generates all combinations of n pairs of well-formed parentheses.
     *
     * @param n the number of pairs of parentheses
     * @return a list of all possible combinations of n pairs of well-formed parentheses
     */
    public List<String> generateParenthesis(int n) {
        this.maxPairs = n;
        // Start the depth-first search with initial values for open and close parentheses count
        generate(0, 0, "");
        return answers;
    }

    /**
     * Helper method to generate the parentheses using depth-first search.
     *
     * @param openCount  the current number of open parentheses
     * @param closeCount the current number of close parentheses
     * @param currentString the current combination of parentheses being built
     */
    private void generate(int openCount, int closeCount, String currentString) {
        // Check if the current counts of open or close parentheses exceed maxPairs or if closeCount exceeds openCount
        if (openCount > maxPairs || closeCount > maxPairs || openCount < closeCount) {
            // The current combination is invalid, backtrack from this path
            return;
        }
        // Check if the current combination is a valid complete set of parentheses
        if (openCount == maxPairs && closeCount == maxPairs) {
            // Add the valid combination to the list of answers
            answers.add(currentString);
            return;
        }
        // Explore the possibility of adding an open parenthesis
        generate(openCount + 1, closeCount, currentString + "(");
        // Explore the possibility of adding a close parenthesis
        generate(openCount, closeCount + 1, currentString + ")");
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to generate all combinations of well-formed parentheses.
    vector<string> generateParenthesis(int n) {
        vector<string> result; // This will store the valid combinations.

        // Use a lambda function to perform depth-first search.
        // 'leftCount' and 'rightCount' track the count of '(' and ')' used respectively.
        // 'current' is the current combination of parentheses.
        function<void(int, int, string)> depthFirstSearch = [&](int leftCount, int rightCount, string current) {
            // If the current combination is invalid (more ')' than '(' or counts exceed 'n'), stop exploration.
            if (leftCount > n || rightCount > n || leftCount < rightCount) return;

            // If the combination is valid and complete, add it to the result list.
            if (leftCount == n && rightCount == n) {
                result.push_back(current);
                return;
            }

            // If we can add a '(', do so and continue the search.
            depthFirstSearch(leftCount + 1, rightCount, current + "(");

            // If we can add a ')', do so and continue the search.
            depthFirstSearch(leftCount, rightCount + 1, current + ")");
        };

        // Start the search with zero counts and an empty combination.
        depthFirstSearch(0, 0, "");

        // Return all the valid combinations found.
        return result;
    }
};

```

Typescript Solution

```typescript
function generateParenthesis(n: number): string[] {
    // Define the result array to store valid combinations of parentheses.
    let result: string[] = [];

    // Define a depth-first search function to explore all possible combinations of parentheses.
    // l: count of left parentheses used, r: count of right parentheses used, currentString: current combination of parentheses
    function depthFirstSearch(leftCount: number, rightCount: number, currentString: string): void {
        // If the number of left or right parentheses exceeds n, or if the number of right parentheses
        // is greater than the left at any point, the current string is invalid.
        if (leftCount > n || rightCount > n || leftCount < rightCount) {
            return;
        }

        // If the current string uses all left and right parentheses correctly, add it to the result.
        if (leftCount === n && rightCount === n) {
            result.push(currentString);
            return;
        }

        // Explore further by adding a left parenthesis if it does not exceed the limit.
        depthFirstSearch(leftCount + 1, rightCount, currentString + '(');

        // Explore further by adding a right parenthesis if it does not exceed the limit.
        depthFirstSearch(leftCount, rightCount + 1, currentString + ')');
    }

    // Start the depth-first search with a count of 0 for both left and right parentheses and an empty string.
    depthFirstSearch(0, 0, '');

    // Return the array of valid combinations.
    return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(4^n / sqrt(n))`. This complexity arises because each valid combination can be represented by a path in a decision tree, which has `2n` levels (since we make a decision at each level to add either a left or a right parenthesis, and we do this `n` times for each parenthesis type). However, not all paths in the tree are valid; the number of valid paths follows the nth Catalan number, which is proportional to `4^n / (n * sqrt(n))`, and `n` is a factor that represents the polynomial part that gets smaller as `n` gets larger. Since we're looking at big-O notation, we simplify this to `4^n / sqrt(n)` for large `n`.

Space Complexity

The space complexity is `O(n)` because the depth of the recursive call stack is proportional to the number of parentheses to generate, which is `2n`, and the space required to store a single generated set of parentheses is also linear to `n`. Hence, the complexity due to the call stack is `O(n)`. The space used to store the answers is separate and does not affect the complexity from a big-O perspective. Keep in mind that the returned list itself will contain `O(4^n / sqrt(n))` elements, and if you consider the space for the output list, the overall space complexity would be `O(n * 4^n / sqrt(n))`, which includes the length of each string times the number of valid strings. Typically, the space complexity considers only the additional space required, not the space for the output. Therefore, we only consider the `O(n)` space used by the call stack for our space complexity analysis.