22. Generate Parentheses

**Dynamic Programming Backtracking** 

5. We continue this process until we have used all n pairs of parentheses.

explore all valid combinations and store them in the list ans.

t: The current combination string formed by adding parentheses.

appends "(" to the current string t.

formed parentheses combinations.

generateParenthesis function.

**Step 2: First Level Recursive Calls** 

**Step 3: Second Level Recursive Calls** 

**Step 5: Fourth Level Recursive Calls** 

"(())" is added to our answer set ans.

**Example Walkthrough** 

**Problem Description** 

String

Medium

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.
- 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 (1
- 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

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).

## Initial Call: The generateParenthesis function initiates the process by calling the nested dfs (Depth First Search) function

**Solution Approach** 

with initial values of zero used opening parentheses (1), zero used closing parentheses (r), and an empty string for the current combination (t).

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

• 1: The number of opening parentheses used so far. r: The number of closing parentheses used so far.

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

Base Case: The recursion has two base cases within the DFS function: a. Invalid Condition: When the number of used opening parentheses 1 is more than n, or the closing parentheses r is more than n or more than 1, it indicates an incorrect combination. The function returns without doing anything. b. Valid Combination: When both 1 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.

Adding an opening parenthesis: If not all n opening parentheses have been used (1 < n), the dfs function calls itself with 1 + 1, r, and</li>

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

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. 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-

that there are some unmatched opening parentheses. Thus, the dfs function calls itself with l, r + 1, and appends ")" to t.

Adding a closing parenthesis: If the number of closing parentheses used is less than the number of opening parentheses (r < 1), it implies</li>

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.

Return Result: Finally, once all possible combinations have been explored, the ans list is returned as the result of the

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).

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

At this stage, we have two choices: add an opening parenthesis or add a closing parenthesis. Since 1 < n, we can add an opening parenthesis. We cannot add a closing parenthesis yet because r < 1 is not satisfied (both 1 and r are 0). So, our recursive calls are:

## parenthesis. The string is now "(".

dfs(2, 1, "(()")

dfs(2, 2, "(())")

dfs(1, 0, "(")

**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

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

The algorithm will backtrack now and explore other paths, but since n = 2 and we have used all our opening parentheses, there

anymore (2 is not less than 2). We must add a closing parenthesis now since r < 1 is satisfied. We get:

After the first opening parenthesis is added, we are again at a stage where we can choose to add an opening or closing

**Step 6: Base Case Reached** Now 1 = 2 and r = 2, which equals n. We have reached a base case where we have a well-formed combination. This combination

• For 1 < 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 1 (r < 1 is not true).

Backtracking

# The completed list ans, now containing "(())", is returned.

are no more paths to discover.

**Step 7: Return Result** 

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 < 1.

# If there are more open or more close parens than 'n', or more close parens than open, it's invalid

Considering another branch of this example, if we go back to the second level again and instead of adding another opening

## • 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 Implementation

return

return

combinations = []

backtrack(0, 0, '')

return combinations

this.maxPairs = n;

generate(0, 0, "");

return answers;

return;

class Solution: def generateParenthesis(self, n: int) -> List[str]: # Helper function for depth-first search def backtrack(open\_count, close\_count, path):

# When the current path uses all parens correctly, add the combination to the results

if open\_count > n or close\_count > n or open\_count < close\_count:</pre>

# Start the recursive search with initial counts of open and close parentheses

\* @return a list of all possible combinations of n pairs of well-formed parentheses

\* Helper method to generate the parentheses using depth-first search.

\* @param currentString the current combination of parentheses being built

private void generate(int openCount, int closeCount, String currentString) {

// The current combination is invalid, backtrack from this path

// Check if the current combination is a valid complete set of parentheses

if (openCount > maxPairs || closeCount > maxPairs || openCount < closeCount) {</pre>

\* @param openCount the current number of open parentheses

\* @param closeCount the current number of close parentheses

if (openCount == maxPairs && closeCount == maxPairs) {

answers.add(currentString);

// Add the valid combination to the list of answers

// Start the depth-first search with initial values for open and close parentheses count

# Continue the search by adding an open paren if possible

# Continue the search by adding a close paren if possible

backtrack(open\_count + 1, close\_count, path + '(')

backtrack(open\_count, close\_count + 1, path + ')')

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

if open\_count == n and close\_count == n:

# This list will hold all the valid combinations

combinations.append(path)

# Return all the valid combinations found

public List<String> generateParenthesis(int n) {

```
/**
* Generates all combinations of n pairs of well-formed parentheses.
* @param n the number of pairs of parentheses
```

\*/

/\*\*

\*/

Java

class Solution {

```
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 + ")");
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;</pre>
            // 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
function generateParenthesis(n: number): string[] {
    // Define the result array to store valid combinations of parentheses.
```

// l: count of left parentheses used, r: count of right parentheses used, currentString: current combination of parentheses

// Define a depth-first search function to explore all possible combinations of parentheses.

// Explore further by adding a left parenthesis if it does not exceed the limit.

// Explore further by adding a right parenthesis if it does not exceed the limit.

// is greater than the left at any point, the current string is invalid.

if (leftCount > n || rightCount > n || leftCount < rightCount) {</pre>

depthFirstSearch(leftCount + 1, rightCount, currentString + '(');

depthFirstSearch(leftCount, rightCount + 1, currentString + ')');

if open\_count > n or close\_count > n or open\_count < close\_count:</pre>

if (leftCount === n && rightCount === n) {

result.push(currentString);

def generateParenthesis(self, n: int) -> List[str]:

def backtrack(open\_count, close\_count, path):

if open\_count == n and close\_count == n:

# Helper function for depth-first search

combinations.append(path)

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

// If the current string uses all left and right parentheses correctly, add it to the result.

// Start the depth-first search with a count of 0 for both left and right parentheses and an empty string.

# If there are more open or more close parens than 'n', or more close parens than open, it's invalid

# When the current path uses all parens correctly, add the combination to the results

// Check if the current counts of open or close parentheses exceed maxPairs or if closeCount exceeds openCount

```
// Return the array of valid combinations.
      return result;
class Solution:
```

let result: string[] = [];

return;

return;

depthFirstSearch(0, 0, '');

return

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
Time and Space Complexity
Time Complexity
  The time complexity of the given code is 0(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<sup>n</sup> / 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 0(4<sup>n</sup> / sqrt(n)) elements, and if you consider the

space for the output list, the overall space complexity would be  $0(n * 4^n / sqrt(n))$ , 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.