<https://leetcode.com/problems/generate-parentheses/>

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

**Example 1:**

Input: n = 3

Output: ["((()))","(()())","(())()","()(())","()()()"]

**Example 2:**

Input: n = 1

Output: ["()"]

**Constraints:**

1 <= n <= 8

**Attempt 1: 2022-11-26**

**Solution 1:  Backtracking**

**Style 1: Decreasing to condition open = close = 0 with StringBuilder + explicit backtracking (10 min)**

class Solution {

    public List<String> generateParenthesis(int n) {

        List<String> result = new ArrayList<String>();

        helper(result, n, n, new StringBuilder());

        return result;

    }

    private void helper(List<String> result, int open, int close, StringBuilder sb) {

        if(open == 0 && close == 0) {

            result.add(sb.toString());

            return;

        }

        if(open > 0) {

            sb.append("(");

            helper(result, open - 1, close, sb);

            sb.setLength(sb.length() - 1);

        }

        if(close > open) {

            sb.append(")");

            helper(result, open, close - 1, sb);

            sb.setLength(sb.length() - 1);

        }

    }

}

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.

If don't understand how "Catalan number" works, we may answer as O(4^n)

Refer to

https://leetcode.com/problems/generate-parentheses/solutions/10100/easy-to-understand-java-backtracking-solution/comments/345076

I think it should be O(2^(2n)). Space is O(n). We need n pairs which means we need n left parenthesis

and n right parenthesis, making it total 2n.

Then we need to decide will we include left or right? Think about the subset problem. We need to answer

it 2^(2n) times.

Some people may argue it O(2^n), I would say the power 2 here is very significant and should not be ignored.

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.

**Style 2: Increasing to condition open = close = n with StringBuilder + explicit backtracking (10 min)**

class Solution {

    public List<String> generateParenthesis(int n) {

        List<String> result = new ArrayList<String>();

        helper(result, 0, 0, n, new StringBuilder());

        return result;

    }

    private void helper(List<String> result, int open, int close, int n, StringBuilder sb) {

        if(open == n && close == n) {

            result.add(sb.toString());

            return;

        }

        if(open < n) {

            sb.append("(");

            helper(result, open + 1, close, n, sb);

            sb.setLength(sb.length() - 1);

        }

        if(close < open) {

            sb.append(")");

            helper(result, open, close + 1, n, sb);

            sb.setLength(sb.length() - 1);

        }

    }

}

**Refer to**

<https://leetcode.com/problems/generate-parentheses/solutions/10100/easy-to-understand-java-backtracking-solution/>

**The idea here is to only add '(' and ')' that we know will guarantee us a solution (instead of adding 1 too many close). Once we add a '(' we will then discard it and try a ')' which can only close a valid '('. Each of these steps are recursively called.**

<https://leetcode.com/problems/generate-parentheses/discuss/10100/Easy-to-understand-Java-backtracking-solution/10993>

Same idea, but instead of expensive string concatenation, I used StringBuilder and backtrack when necessary, mainly for efficiency.

**Style 3: Use String without explicit backtracking statement (10 min)**

**The difference between StringBuilder keeps the value to next iteration and String works like primitive type and keeps original value as immutable**

Because when using simple String, you pass str+"(" or str+")" to the next function by keeping the str the original value. However, when using the StringBuilder, you plus "(" or ")" to the str first, then you pass the str to the next function. In this case, the str becomes a new value. Thus, you have to delete them after the function called so that when it comes to the next "if" statement, the str will stay in the original value.

class Solution {

    public List<String> generateParenthesis(int n) {

        List<String> result = new ArrayList<String>();

        helper(result, n, n, "");

        return result;

    }

    private void helper(List<String> result, int open, int close, String str) {

        if(open == 0 && close == 0) {

            result.add(str);

            return;

        }

        if(open > 0) {

            helper(result, open - 1, close, str + "(");

        }

        if(close > open) {

            helper(result, open, close - 1, str + ")");

        }

    }

}

----------------------------------------------------------------------------------

OR the reverse version

----------------------------------------------------------------------------------

class Solution {

    public List<String> generateParenthesis(int n) {

        List<String> result = new ArrayList<String>();

        helper(result, 0, 0, n, "");

        return result;

    }

    private void helper(List<String> result, int open, int close, int n, String str) {

        if(open == n && close == n) {

            result.add(str);

            return;

        }

        if(open < n) {

            helper(result, open + 1, close, n, str + "(");

        }

        if(close < open) {

            helper(result, open, close + 1, n, str + ")");

        }

    }

}

**Style 4: Another use String without explicit backtracking statement (10 min)**

class Solution {

    // List to hold all the valid parentheses combinations

    private List<String> answers = new ArrayList<>();

    /\*\*

     \* 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) {

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

        generate(0, 0, "", n);

        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, int maxPairs) {

        // 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 + "(", maxPairs);

        // Explore the possibility of adding a close parenthesis

        generate(openCount, closeCount + 1, currentString + ")", maxPairs);

    }

}

**Refer to**

<https://algo.monster/liteproblems/22>

**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.

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 ")".

**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.**

**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.**

**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.**

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

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:

**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).

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.

**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.

**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.

**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.

**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.**

**这里的 Backtracking 是典型的 implicit backtracking，简单说就是利用了DFS的本质：一直递归到底的backtracking，因为到了底 (base return condition) 的时候就会自然返回上一层，实际作用等价于 backtracking, 而 explicit backtracking 一般在递归到底之前就被提前激活了，相当于节约了递归的时间和空间，当然，是否使用 explicit backtracking 或者 implicit backtracking 也基于递归过程中使用的承载字符串的是 StringBuilder 还是 String，如果是 StringBuilder 就需要 explicit backtracking 来手动恢复成上一层递归的状态，如果是 String 就不需要手动恢复成上一层的递归状态（基于 String immutable 的原理）**

**Refer to**

<https://leetcode.com/problems/generate-parentheses/solutions/10100/easy-to-understand-java-backtracking-solution/comments/531279>

If you change the input parameter type String to StringBuilder as follow, it is explicit the solution is a typical backtracking, which has a obvious stepping back sb.deleteCharAt(sb.length() - 1);. I think it is String type makes backtracking implicit and lets people consider it a common DFS solution. In Java, String is a final type, changing a string will generate a new string, which disables one string's ability in memorialized searching (compared to one StringBuilder) in multilevel function calls. Thus in the String appoach, through generating new string (by appending a parentheses to the string int the previous function call) for memorialized searching, we don't need an explicit stepping back like sb.deleteCharAt(sb.length() - 1);.

**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 "()()".

**Solution Implementation**

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 + ")");

}

}

**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.

**Refer to Deepseek**

Here's a Java solution using backtracking to generate all combinations of well-formed parentheses:

public class Solution {

public List<String> generateParenthesis(int n) {

List<String> result = new ArrayList<>();

backtrack(result, "", 0, 0, n);

return result;

}

private void backtrack(List<String> result, String current, int open, int close, int max) {

if (current.length() == max \* 2) {

result.add(current);

return;

}

if (open < max) {

backtrack(result, current + "(", open + 1, close, max);

}

if (close < open) {

backtrack(result, current + ")", open, close + 1, max);

}

}

}

**Explanation:**

**Base Case**: When the current string's length reaches 2\*n (since each pair has 2 parentheses), we add it to the result list.

**Recursive Cases**:

Add an opening parenthesis '(' if the count of opening parentheses is less than n

Add a closing parenthesis ')' if the count of closing parentheses is less than opening parentheses

**Backtracking**: The algorithm builds the solution incrementally by trying all valid possibilities at each step, and backtracking when it hits an invalid state (which is implicitly handled by the conditions).

**Time and Space Complexity:**

**Time Complexity**: O(4^n/√n) - This is the nth Catalan number which is the exact number of valid combinations.

**Space Complexity**: O(4^n/√n) for storing all combinations, and O(n) for the recursion stack.

This approach efficiently generates all valid combinations by ensuring at each step that we never have more closing parentheses than opening ones.

**I didn't see explicit backtracking statement, is that because we use String type parameter in recursion call ? And if yes, is that means if we use StringBuilder type parameter in recursion call, we have to write explicit backtracking statement ?**

Yes, your understanding is correct. Let me clarify the difference:

**1. Using String (Original Solution) - Implicit Backtracking**

backtrack(result, current + "(", ...);

**Immutable Strings**: Each recursive call gets a **new copy** of the string

**Automatic State Revert**: When recursion returns to parent call, the original current remains unchanged

**No cleanup needed**: The parameter is passed by value, so modifications stay in child calls

**2. Using StringBuilder - Explicit Backtracking Required**

// With mutable StringBuilder

current.append('(');

backtrack(...);

current.deleteCharAt(current.length() - 1); // REQUIRED CLEANUP

**Mutable Object**: All recursive calls share the **same** StringBuilder instance

**Explicit Backtracking**: Must manually remove added characters after recursion

**Why?**: Otherwise previous calls would retain modifications from deeper calls

**Modified Solution with StringBuilder (Explicit Backtracking):**

public class Solution {

public List<String> generateParenthesis(int n) {

List<String> result = new ArrayList<>();

backtrack(result, new StringBuilder(), 0, 0, n);

return result;

}

private void backtrack(List<String> result, StringBuilder sb, int open, int close, int max) {

// Base case: valid combination found

if (sb.length() == max \* 2) {

result.add(sb.toString());

return;

}

// Case 1: Add '(' if possible

if (open < max) {

sb.append('(');

backtrack(result, sb, open + 1, close, max);

sb.deleteCharAt(sb.length() - 1); // Explicit backtracking

}

// Case 2: Add ')' if valid

if (close < open) {

sb.append(')');

backtrack(result, sb, open, close + 1, max);

sb.deleteCharAt(sb.length() - 1); // Explicit backtracking

}

}

}

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **String Version** | **StringBuilder Version** |
| **Mutability** | Immutable | Mutable |
| **Memory Efficiency** | Creates new strings (higher overhead) | Modifies in-place (more efficient) |
| **Backtracking** | Automatic (no code needed) | Manual  deleteCharAt()  required |
| **Parameter Passing** | Pass-by-value (safe) | Pass-by-reference (shared state) |
| **Recursion State** | Each call has isolated state | All calls share same object |

**Why This Matters:**

**String**: Simpler, less error-prone, but creates many temporary strings

**StringBuilder**: More efficient (O(n) vs O(n²) per recursive call), but requires careful state management

**Best Practice**: Use String for simplicity in small inputs (like LeetCode), StringBuilder for memory-critical applications

Both solutions have the same time complexity (O(4^n/√n)), but the StringBuilder version reduces memory allocation overhead by avoiding creating new string objects at each recursion level.

**Refer to**

[L20.P11.5.Valid Parentheses (Ref.L32)](note://8CA124C3F0314C4A9CC7078E4E81EB04)