<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 (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);

}

}

}

**Style 2: Increasing to condition open = close = n with StringBuilder (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/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 backtrack 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 + ")");

}

}

}