**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

/\*\*

\* @param {number} n

\* @return {string[]}

\*/

var generateParenthesis = function(n) {

};

Approach 1: Brute Force

**Intuition**

We can generate all 2^{2n}22*n* sequences of '(' and ')' characters. Then, we will check if each one is valid.

**Algorithm**

To generate all sequences, we use a recursion. All sequences of length n is just '(' plus all sequences of length n-1, and then ')' plus all sequences of length n-1.

To check whether a sequence is valid, we keep track of balance, the net number of opening brackets minus closing brackets. If it falls below zero at any time, or doesn't end in zero, the sequence is invalid - otherwise it is valid.

class Solution {

public List<String> generateParenthesis(int n) {

List<String> combinations = new ArrayList();

generateAll(new char[2 \* n], 0, combinations);

return combinations;

}

public void generateAll(char[] current, int pos, List<String> result) {

if (pos == current.length) {

if (valid(current))

result.add(new String(current));

} else {

current[pos] = '(';

generateAll(current, pos+1, result);

current[pos] = ')';

generateAll(current, pos+1, result);

}

}

public boolean valid(char[] current) {

int balance = 0;

for (char c: current) {

if (c == '(') balance++;

else balance--;

if (balance < 0) return false;

}

return (balance == 0);

}

}

**Complexity Analysis**

* Time Complexity : O(2^{2n}n)*O*(22*nn*). For each of 2^{2n}22*n* sequences, we need to create and validate the sequence, which takes O(n)*O*(*n*) work.
* Space Complexity : O(2^{2n}n)*O*(22*nn*). Naively, every sequence could be valid. See [Approach 3](https://leetcode.com/problems/generate-parentheses/solution/#approach-3-closure-number) for development of a tighter asymptotic bound.

#### Approach 2: Backtracking

**Intuition and Algorithm**

Instead of adding '(' or ')' every time as in [Approach 1](https://leetcode.com/problems/generate-parentheses/solution/#approach-1-brute-force), let's only add them when we know it will remain a valid sequence. We can do this by keeping track of the number of opening and closing brackets we have placed so far.

We can start an opening bracket if we still have one (of n) left to place. And we can start a closing bracket if it would not exceed the number of opening brackets.

class Solution {

public List<String> generateParenthesis(int n) {

List<String> ans = new ArrayList();

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

return ans;

}

public void backtrack(List<String> ans, StringBuilder cur, int open, int close, int max){

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

ans.add(cur.toString());

return;

}

if (open < max) {

cur.append("(");

backtrack(ans, cur, open+1, close, max);

cur.deleteCharAt(cur.length() - 1);

}

if (close < open) {

cur.append(")");

backtrack(ans, cur, open, close+1, max);

cur.deleteCharAt(cur.length() - 1);

}

}

}

**Complexity Analysis**

Our complexity analysis rests on understanding how many elements there are in generateParenthesis(n). This analysis is outside the scope of this article, but it turns out this is the n-th Catalan number \dfrac{1}{n+1}\binom{2n}{n}*n*+11​(*n*2*n*​), which is bounded asymptotically by \dfrac{4^n}{n\sqrt{n}}*nn*​4*n*​.

* Time Complexity : O(\dfrac{4^n}{\sqrt{n}})*O*(*n*​4*n*​). Each valid sequence has at most n steps during the backtracking procedure.
* Space Complexity : O(\dfrac{4^n}{\sqrt{n}})*O*(*n*​4*n*​), as described above, and using O(n)*O*(*n*) space to store the sequence.

Approach 3: Closure Number

**Intuition**

To enumerate something, generally we would like to express it as a sum of disjoint subsets that are easier to count.

Consider the *closure number* of a valid parentheses sequence S: the least index >= 0 so that S[0], S[1], ..., S[2\*index+1] is valid. Clearly, every parentheses sequence has a unique *closure number*. We can try to enumerate them individually.

**Algorithm**

For each closure number c, we know the starting and ending brackets must be at index 0 and 2\*c + 1. Then, the 2\*c elements between must be a valid sequence, plus the rest of the elements must be a valid sequence.

class Solution {

public List<String> generateParenthesis(int n) {

List<String> ans = new ArrayList();

if (n == 0) {

ans.add("");

} else {

for (int c = 0; c < n; ++c)

for (String left: generateParenthesis(c))

for (String right: generateParenthesis(n-1-c))

ans.add("(" + left + ")" + right);

}

return ans;

}

}

**Complexity Analysis**

* Time and Space Complexity : O(\dfrac{4^n}{\sqrt{n}})*O*(*n*​4*n*​). The analysis is similar to [Approach 2](https://leetcode.com/problems/generate-parentheses/solution/#approach-2-backtracking).