package com.iimtiaz.day\_13;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class GenerateParentheses {  
 public static void main(String[] args) {  
 System.*out*.println(Arrays.*asList*(new Solution().generateParenthesis(3)));  
 }  
}  
  
*/\*\*  
 \* Time Complexity: O(2^n)  
 \* <p>  
 \* Exponential Growth: The recursive calls create a branching tree of possible combinations, leading to exponential  
 \* growth in the number of function calls.  
 \* Each Call: Each individual call involves constant-time operations, but the total number of calls grows exponentially  
 \* with n.  
 \* Space Complexity: O(n)  
 \* <p>  
 \* Recursion Stack: The recursion stack can grow up to a depth of n in the worst case, storing intermediate states of  
 \* the backtracking process.  
 \* Result List: The result list holds valid combinations, but its size is bounded by O(2^n) and is generally considered  
 \* smaller than the recursion stack's impact on space complexity.  
 \*/*class Solution {  
 public List generateParenthesis(int n) {  
 List result = new ArrayList<>();  
 findAll("", result, 0, 0, n);  
 return result;  
 }  
  
 void findAll(String current, List<String> result, int op, int cl, int n) {  
 if (current.length() == 2 \* n) {  
 result.add(current);  
 return;  
 }  
 if (op < n) {  
 findAll(current + "(", result, op + 1, cl, n);  
 }  
 if (op > cl) {  
 findAll(current + ")", result, op, cl + 1, n);  
 }  
 }  
}  
  
*/\*\*  
 \* Here’s a breakdown of the code:  
  
 \* generateParenthesis(int n): This is the main function that initializes the result list and calls the findAll function  
 \* to start the process. The parameter n is the number of pairs of parentheses.  
 \* <p>  
 \* findAll(String current, List<String> result, int op, int cl, int n): This is a recursive function that generates all  
 \* possible combinations of parentheses. It uses backtracking to add an open parenthesis “(” or a close parenthesis “)”  
 \* to the current combination current and then recursively calls itself until a valid combination is found.  
 \* <p>  
 \* current: The current combination of parentheses.  
 \* result: The list to store all valid combinations of parentheses.  
 \* op: The number of open parentheses used so far.  
 \* cl: The number of close parentheses used so far.  
 \* n: The total number of pairs of parentheses.  
 \* The function checks the following conditions:  
 \* <p>  
 \* If the length of current is equal to 2\*n, it means we have a valid combination of n pairs of parentheses, so we add  
 \* current to the result list.  
 \* If the number of open parentheses op used so far is less than n, we can add an open parenthesis to current and  
 \* recursively call findAll.  
 \* If the number of open parentheses op used so far is more than the number of close parentheses cl, we can add a close  
 \* parenthesis to current and recursively call findAll.  
 \* The logic behind this solution is to generate all possible combinations of parentheses and only add the valid ones to  
 \* the result list. The validity of a combination is ensured by the condition that a close parenthesis can only be added  
 \* if there are more open parentheses than close parentheses in the current combination.  
 \*/*// https://leetcode.com/problems/generate-parentheses/