**Саурон и многомерное Кольцо Всевластья**

*Оказывается, Саурон, некогда правивший всем Мордором, развоплотился не навсегда.*

*Он вновь готовит легионы орков, а Ородруин непрерывно засыпает все окрестности чёрным пеплом. Единственный способ противостоять растущей власти Саурона -- изобрести заклинание генерации многомерного Кольца Всевластья, дабы полностью и окончательно развоплотить Саурона по всему Иномирью.*

Напишите рекурсивную функцию, которая выдаёт все корректные сбалансированные комбинации круглых скобок.

На вход функции поступает количество открывающих скобок.

Например,

1 = "()"

2 = "(())" "()()"

Внешне и внутренне несбалансированные комбинации считаются ошибочными:

"()))" ")(()" "(()" -- неверно

Функция

string BalancedParentheses(int N)

получает на вход количество открывающих скобок N (N >= 1) и возвращает строку, в которой через пробел записаны все сгенерированные комбинации скобок (в произвольном порядке).

Например,

BalancedParentheses(1) = "()"

BalancedParentheses(2) = "(()) ()()"

[как постить решение](https://skillsmart.ru/algo/lvl1/how.html)

6,4

6,3,1

6,2,2

6,2,1,1

6,1,1,1,1

int z1 = z;  
  
while (z1 > 1) {  
  
 int len = *brackets*.size() - 1;  
  
 for (int l = len; l > 0; l--) {  
  
 if (*brackets*.get(l) > 1) {  
 System.*out*.println("reducing " + *brackets*.get(l));  
  
 z1 = *reduce*(*brackets*.get(l)).get(0);  
 System.*out*.println("z1 reduced to " + z1);  
  
 if (z1 > 1 && (x - (y + z1) >= z1)){  
 int sum1 = y + z1;  
 *brackets*.clear();  
 *brackets*.add(y);  
 while (sum1 <= x){  
 *brackets*.add(z1);  
 sum1 = sum1 + z1;  
 }  
 System.*out*.println(*brackets*);  
  
 }  
  
 *brackets*.add(*reduce*(*brackets*.get(l)).get(0));  
 *brackets*.add(*reduce*(*brackets*.get(l)).get(1));  
 *brackets*.remove(l);  
  
  
  
  
 System.*out*.println(*brackets*);  
 }  
 }  
  
  
  
}

ниже промежуточное решение с двумя реализованными методами:

cascade – сравнивает два самых правых числа, и если их разница > 1 выполняет т.н. «каскад»: 4,3,1 ---> 4,2,2

reduce – уменьшает число N , разделяя его на N-1 и 1

4,2,2 --> 4,2,1,1

import java.util.ArrayList;  
import java.util.LinkedList;  
import java.util.List;  
  
public class Brackets {  
  
 static LinkedList<Integer> *brackets* = new LinkedList<>();  
 public static final int *x* = 10;  
  
 public static String BalancedParentheses(int N){  
 String output = "output";  
  
 return output;  
 }  
  
 public static void reduceLast(int value, int index){  
 if (value > 1){  
 *brackets*.add(index, 1);  
 *brackets*.add(index, value - 1);  
 }  
 }  
  
  
 public static void reduceCascade(LinkedList<Integer> inputList, int index){  
 LinkedList<Integer> tempList = new LinkedList<>();  
 for (int i = 0; i <= index; i++){  
 tempList.add(inputList.get(i));  
 }  
 int a = tempList.getFirst();  
 int cap = tempList.getLast() - 1;  
  
 System.*out*.println("~~~~~~~~~~~~~~~~~~~");  
 System.*out*.println("preforming cascade on " + inputList);  
 System.*out*.println("a = " + a + ", cap = " + cap);  
  
 tempList.removeLast();  
 tempList.add(cap);  
 System.*out*.println("workinglist: " + tempList);  
  
 if ((a + cap) < *x*){  
 int sum = 0;  
 for (int j = 0; j < tempList.size(); j++){  
 sum = sum + tempList.get(j);  
 }  
 int gap1 = *x* - sum;  
 System.*out*.println("initial gap = " + gap1);  
 while (gap1 >= cap) {  
 tempList.add(cap);  
 gap1 -= cap;  
 //System.out.println("added cap " + cap + ", gap1 reduced to: " + gap1);  
 }  
 if (gap1 != 0 && gap1 < cap) {  
 System.*out*.println("gap1 < cap");  
 tempList.add(gap1);  
 }  
 }  
 System.*out*.println("after cascade: " + tempList);  
 System.*out*.println("~~~~~~~~~~~~~~~~~~~");  
  
 *brackets*.clear();  
 *brackets* = (LinkedList<Integer>) tempList.clone();  
  
 }  
  
 public static void main(String[] args) {  
  
  
 int z = 0;  
 int y = *x* - z;  
 int cap = 0;  
  
 while (y > 1){  
 System.*out*.println("\ncurrent y = " + y + ", z = " + z);  
  
 if (z == 0) {  
 *brackets*.add(y);  
 System.*out*.println(*brackets*);  
 }  
  
 if (z == 1){  
 *brackets*.add(y);  
 *brackets*.add(z);  
 System.*out*.println(*brackets*);  
 }  
  
 if (z > 1) {  
 *brackets*.add(y);  
 *brackets*.add(z);  
  
 if ((y + z) < *x*) {  
 int gap = *x* - (y + z);  
 while (gap >= y) {  
 *brackets*.add(y);  
 gap -= y;  
 }  
 if (gap != 0) {  
 *brackets*.add(gap);  
 }  
 }  
 System.*out*.println(*brackets*);  
 //next y list created  
  
 while (*brackets*.get(1) > 1) {  
  
 for (int r = *brackets*.size() - 1; r > 0; r--) {  
  
 if (r < *brackets*.size() - 1 && (*brackets*.get(r) - *brackets*.get(r + 1) > 1 )){  
 *reduceCascade*(*brackets*, r);  
 break;  
 }  
  
 if (*brackets*.get(r) > 1) {  
 // System.out.println("reduce index = " + r + ", value = " + brackets.get(r));  
 *reduceLast*(*brackets*.get(r), r);  
 *brackets*.remove(r + 2);  
 System.*out*.println(*brackets*);  
 break;  
 }  
 }  
  
  
 }  
 // System.out.println(brackets);  
  
 }  
  
  
 y--;  
 if (z < y) {  
 z++;  
 }  
 else z = y;  
 *brackets*.clear();  
 }  
  
  
  
 if (y == 1){  
 for (int i = 0; i < *x*; i++)  
 *brackets*.add(1);  
 System.*out*.println("\nLast row");  
 System.*out*.println(*brackets*);  
 }  
  
 }  
}

осталось дописать итерацию по каждой строке (легко! :))))

и преобразовать все это дело в скобки (ваще ерунда:))))))

перебор всех вариантов списка (изначально отсортированного по убыванию)

внизу полностью рабочее решение

последние два метода – преобразования из списка чисел в пары скобок с добавлением в финальную строку

import java.util.ArrayList;  
import java.util.\*;  
import java.util.List;  
  
public class Brackets {  
  
  
 public static String BalancedParentheses(int N){  
  
 LinkedList<Integer> brackets = new LinkedList<>();  
 LinkedList<Integer> bracketsClone;  
  
 int x = N;  
 int z = 0;  
 int y = x - z;  
  
 LinkedList<LinkedList<Integer>> finalList = new LinkedList<>();  
 LinkedList<LinkedList<Integer>> testList3 ;  
 LinkedList<LinkedList<Integer>> testList4 ;  
  
 while (y > 1){  
  
 if (z == 0) {  
 brackets.add(y);  
  
 bracketsClone = (LinkedList<Integer>) brackets.clone();  
 finalList.add(bracketsClone);  
 }  
  
 if (z == 1){  
 brackets.add(y);  
 brackets.add(z);  
  
 bracketsClone = (LinkedList<Integer>) brackets.clone();  
 finalList.add(*listPermutations*(bracketsClone).get(0));  
 bracketsClone = (LinkedList<Integer>) brackets.clone();  
 finalList.add(*listPermutations*(bracketsClone).get(1));  
 }  
  
 if (z > 1) {  
 brackets.add(y);  
 brackets.add(z);  
  
 if ((y + z) < x) {  
 int gap = x - (y + z);  
 while (gap >= y) {  
 brackets.add(y);  
 gap -= y;  
 }  
 if (gap != 0) {  
 brackets.add(gap);  
 }  
 }  
  
  
 bracketsClone = (LinkedList<Integer>) brackets.clone();  
 testList3 = *listPermutations*(bracketsClone);  
 testList4 = (*removeDuplicates*(testList3));  
  
 for (int i = 0; i < testList4.size(); i++){  
 finalList.add(testList4.get(i));  
 }  
  
  
  
  
 while (brackets.get(1) > 1) {  
  
 for (int r = brackets.size() - 1; r > 0; r--) {  
  
 if (r < brackets.size() - 1 && (brackets.get(r) - brackets.get(r + 1) > 1 )){  
  
 LinkedList<Integer> tempList2 = *reduceCascade*(brackets, r, x);  
  
  
 brackets.clear();  
 brackets = (LinkedList<Integer>) tempList2.clone();  
  
 bracketsClone = (LinkedList<Integer>) brackets.clone();  
 testList3 = *listPermutations*(bracketsClone);  
 testList4 = (*removeDuplicates*(testList3));  
  
 for (int i = 0; i < testList4.size(); i++){  
 finalList.add(testList4.get(i));  
 }  
  
  
 break;  
 }  
  
 if (brackets.get(r) > 1) {  
 *reduceLast*(brackets, brackets.get(r), r);  
 brackets.remove(r + 2);  
  
 bracketsClone = (LinkedList<Integer>) brackets.clone();  
 testList3 = *listPermutations*(bracketsClone);  
 testList4 = (*removeDuplicates*(testList3));  
  
 for (int i = 0; i < testList4.size(); i++){  
 finalList.add(testList4.get(i));  
 }  
  
  
 break;  
 }  
 }  
  
 }  
  
 }  
  
 y--;  
 if (z < y) {  
 z++;  
 }  
 else z = y;  
 brackets.clear();  
 }  
  
 if (y == 1){  
 for (int i = 0; i < x; i++)  
 brackets.add(1);  
 finalList.add(brackets);  
 }  
  
 //final printouts  
 String output = *bracketGenerator*(finalList);  
 return output;  
 }  
  
 public static LinkedList<Integer> reduceLast(LinkedList<Integer> inputList, int value, int index){  
 if (value > 1){  
 inputList.add(index, 1);  
 inputList.add(index, value - 1);  
 }  
 return inputList;  
 }  
  
 // generating permutations and eliminating duplicates  
 public static LinkedList<LinkedList<Integer>> listPermutations(LinkedList<Integer> list) {  
  
 if (list.size() == 0) {  
 LinkedList<LinkedList<Integer>> result = new LinkedList<>();  
 result.add(new LinkedList<>());  
 return result;  
 }  
  
 LinkedList<LinkedList<Integer>> returnMe = new LinkedList<>();  
 Integer firstElement = list.remove(0);  
  
 LinkedList<LinkedList<Integer>> recursiveReturn = *listPermutations*(list);  
 for (LinkedList<Integer> li : recursiveReturn) {  
  
 for (int index = 0; index <= li.size(); index++) {  
 LinkedList<Integer> temp = new LinkedList<>(li);  
 temp.add(index, firstElement);  
 returnMe.add(temp);  
 }  
 }  
 return returnMe;  
 }  
  
 //removing duplicates (of permutations)  
 public static LinkedList<LinkedList<Integer>> removeDuplicates(LinkedList<LinkedList<Integer>> inputList){  
 LinkedList<LinkedList<Integer>> outputList = new LinkedList<>();  
 for (int i = 0; i < inputList.size(); i++) {  
 if (!outputList.contains(inputList.get(i))){  
 outputList.add(inputList.get(i));  
 }  
 }  
 return outputList;  
 }  
  
 public static LinkedList<Integer> reduceCascade(LinkedList<Integer> inputList, int index, int x){  
 LinkedList<Integer> tempList = new LinkedList<>();  
 for (int i = 0; i <= index; i++){  
 tempList.add(inputList.get(i));  
 }  
 int a = tempList.getFirst();  
 int cap = tempList.getLast() - 1;  
  
 tempList.removeLast();  
 tempList.add(cap);  
  
 if ((a + cap) < x){  
 int sum = 0;  
 for (int j = 0; j < tempList.size(); j++){  
 sum = sum + tempList.get(j);  
 }  
 int gap1 = x - sum;  
 while (gap1 >= cap) {  
 tempList.add(cap);  
 gap1 -= cap;  
 }  
 if (gap1 != 0 && gap1 < cap) {  
 tempList.add(gap1);  
 }  
 }  
 return tempList;  
 }  
  
 public static String bracketGenerator(LinkedList<LinkedList<Integer>> inputList){  
 StringBuilder outputArr = new StringBuilder();  
  
 LinkedList<Integer> subList;  
 for (int i = 0; i < inputList.size(); i++){  
 subList = inputList.get(i);  
  
 for (int j = 0; j < subList.size(); j++){  
 int s = subList.get(j);  
 outputArr.append(*bracketPrinter*(s));  
 }  
 outputArr.append(" ");  
 }  
  
  
 String finalString = outputArr.toString();  
 return finalString;  
  
 }  
  
 public static String bracketPrinter (int b){  
 StringBuilder printer = new StringBuilder();  
  
 for (int i = 0; i < b; i++){  
 printer.append("(");  
 }  
 for (int i = 0; i < b; i++){  
 printer.append(")");  
 }  
  
 String output = printer.toString();  
 return output;  
 }  
  
 public static void main(String[] args) {  
  
 System.*out*.println(*BalancedParentheses*(5));  
  
  
  
 }  
}