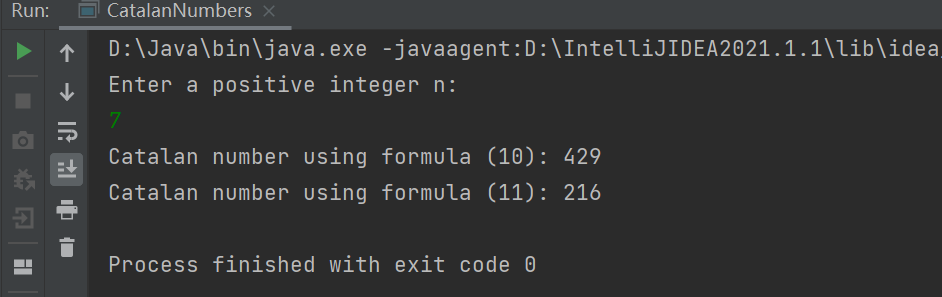
# Computer Projects

1. Given a positive integer n as input, find the Catalan number cn using

a) the recurrence relation (10).

b) the recurrence relation (11).

import java.util.Scanner;  
import java.math.BigInteger;  
  
public class CatalanNumbers {  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter a positive integer n:");  
 int n = sc.nextInt();  
 if (n < 0) {  
 System.*out*.println("n must be a non-negative integer.");  
 return;  
 }  
 System.*out*.println("Catalan number using formula (10): " + *catalan1*(n));  
 System.*out*.println("Catalan number using formula (11): " + *catalan2*(n));  
 }  
  
 public static long catalan1(int n) {  
 long[] catalan = new long[n + 1];  
 catalan[0] = 1;  
 for (int i = 1; i <= n; i++) {  
 catalan[i] = 0;  
 for (int k = 0; k < i; k++) {  
 catalan[i] += catalan[k] \* catalan[i - 1 - k];  
 }  
 }  
 return catalan[n];  
 }  
  
 public static BigInteger catalan2(int n) {  
 BigInteger[] catalan = new BigInteger[n + 1];  
 catalan[0] = BigInteger.*ONE*;  
 for (int i = 1; i <= n; i++) {  
 BigInteger numerator = BigInteger.*valueOf*(4 \* i - 2);  
 BigInteger denominator = BigInteger.*valueOf*(i + 1);  
 catalan[i] = numerator.divide(denominator).multiply(catalan[i - 1]);  
 }  
 return catalan[n];  
 }  
}



2. Given positive integers n and N, find N random sequences xj = {xij} of n

1s and n −1s, compute the corresponding sequences sj = {sij} of partial

sums sij = x1j + x2j + ··· + xij , and let a, b be the number of positive,

nonnegative sequences sj, respectively. Compute the ratios

a) a/C(2n, n).

b) b/C(2n, n).

c) a/b.

import java.math.BigInteger;  
import java.util.\*;  
  
public class CatalanRatios {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("Enter n and N:");  
 int n = scanner.nextInt();  
 int N = scanner.nextInt();  
  
 int a = 0; // Number of sequences with all positive partial sums  
 int b = 0; // Number of sequences with all non-negative partial sums  
  
 for (int i = 0; i < N; i++) {  
 List<Integer> sequence = new ArrayList<>(2 \* n);  
 for (int j = 0; j < n; j++) {  
 sequence.add(1);  
 }  
 for (int j = 0; j < n; j++) {  
 sequence.add(-1);  
 }  
 Collections.*shuffle*(sequence, new Random());  
  
 int sum = 0;  
 boolean allPositive = true;  
 boolean allNonNegative = true;  
 for (int val : sequence) {  
 sum += val;  
 if (sum <= 0) {  
 allPositive = false;  
 }  
 if (sum < 0) {  
 allNonNegative = false;  
 }  
 }  
 if (allPositive) {  
 a++;  
 }  
 if (allNonNegative) {  
 b++;  
 }  
 }  
  
 // Compute C(2n, n)  
 BigInteger c = BigInteger.*ONE*;  
 for (int k = 1; k <= n; k++) {  
 c = c.multiply(BigInteger.*valueOf*(2 \* n - k + 1))  
 .divide(BigInteger.*valueOf*(k));  
 }  
  
 // Compute ratios  
 double ratioA = (double) a / c.doubleValue();  
 double ratioB = (double) b / c.doubleValue();  
 double ratioC = b == 0 ? Double.*POSITIVE\_INFINITY* : (double) a / b;  
  
 // Print results  
 System.*out*.println("a/C(2n, n): " + ratioA);  
 System.*out*.println("b/C(2n, n): " + ratioB);  
 System.*out*.println("a/b: " + ratioC);  
 }  
}

3. Given a positive integer n, find a random sequence x of n 1s and n −1s

that has a nonnegative sequence s of partial sums. Using s, produce the

corresponding

a) well-formed sequence of parentheses.

b) stack permutation.

c) well-parenthesized product.

d) full binary tree (graphics).

e) triangulation of a convex polygon (graphics).

import java.util.\*;  
import java.util.stream.Collectors;  
  
public class DyckPathGenerator {  
  
 public static void main(String[] args) {  
 int n = 3; // Example value for n  
 Random rand = new Random();  
  
 // Generate Dyck path sequence  
 List<Integer> sequence = *generateDyckPath*(n, rand);  
 System.*out*.println("Dyck Path Sequence: " + sequence);  
  
 // a) Well-formed parentheses  
 String parentheses = sequence.stream()  
 .map(i -> i == 1 ? "(" : ")")  
 .collect(Collectors.*joining*());  
 System.*out*.println("Parentheses: " + parentheses);  
  
 // b) Stack permutation  
 Stack<Integer> stack = new Stack<>();  
 List<Integer> permutation = new ArrayList<>();  
 int pushValue = 1;  
 for (int step : sequence) {  
 if (step == 1) {  
 stack.push(pushValue);  
 pushValue++;  
 } else {  
 permutation.add(stack.pop());  
 }  
 }  
 System.*out*.println("Stack Permutation: " + permutation);  
  
 // c) Well-parenthesized product  
 Stack<String> exprStack = new Stack<>();  
 int term = 1;  
 for (int step : sequence) {  
 if (step == 1) {  
 exprStack.push(String.*valueOf*(term));  
 term++;  
 } else {  
 String b = exprStack.pop();  
 String a = exprStack.pop();  
 exprStack.push("(" + a + "\*" + b + ")");  
 }  
 }  
 String product = exprStack.pop();  
 System.*out*.println("Well-parenthesized Product: " + product);  
  
 // d) Full binary tree structure  
 TreeNode root = *buildBinaryTree*(sequence);  
 *printBinaryTree*(root, "");  
 System.*out*.println();  
 }  
  
 // Generate Dyck path sequence  
 private static List<Integer> generateDyckPath(int n, Random rand) {  
 List<Integer> sequence = new ArrayList<>();  
 int currentSum = 0;  
 int remaining1 = n;  
 int remainingNeg1 = n;  
 while (sequence.size() < 2 \* n) {  
 List<Integer> possibleSteps = new ArrayList<>();  
 if (remaining1 > 0) {  
 possibleSteps.add(1);  
 }  
 if (currentSum > 0) {  
 possibleSteps.add(-1);  
 }  
 if (possibleSteps.isEmpty()) {  
 return null; // Invalid state, should not happen with proper constraints  
 }  
 int randomIndex = rand.nextInt(possibleSteps.size());  
 int step = possibleSteps.get(randomIndex);  
 sequence.add(step);  
 if (step == 1) {  
 currentSum++;  
 remaining1--;  
 } else {  
 currentSum--;  
 remainingNeg1--;  
 }  
 }  
 return sequence;  
 }  
  
 // Build full binary tree from Dyck path  
 private static TreeNode buildBinaryTree(List<Integer> sequence) {  
 Stack<TreeNode> treeStack = new Stack<>();  
 TreeNode root = null;  
 int value = 1;  
 for (int step : sequence) {  
 if (step == 1) {  
 TreeNode node = new TreeNode(value);  
 value++;  
 if (treeStack.isEmpty()) {  
 root = node;  
 } else {  
 if (treeStack.peek().left == null) {  
 treeStack.peek().left = node;  
 } else {  
 treeStack.peek().right = node;  
 }  
 }  
 treeStack.push(node);  
 } else {  
 treeStack.pop();  
 }  
 }  
 return root;  
 }  
  
 // Print binary tree structure  
 private static void printBinaryTree(TreeNode node, String indent) {  
 if (node == null) {  
 return;  
 }  
 System.*out*.println(indent + "Node " + node.value);  
 *printBinaryTree*(node.left, indent + " ");  
 *printBinaryTree*(node.right, indent + " ");  
 }  
  
 // TreeNode class for binary tree  
 private static class TreeNode {  
 int value;  
 TreeNode left;  
 TreeNode right;  
  
 TreeNode(int value) {  
 this.value = value;  
 left = null;  
 right = null;  
 }  
 }  
}

