École Polytechnique

Bachelor of Science

Algorithms for Discrete Mathematics

Algorithms for Numerical Analysis of Catalan Numbers

Authors:
Yubo Cai

Supervisor:
Lucas Gerin

January 28, 2023

Abstract

This report is for the presentation of The final project for MAA205 - Algorithms for Discrete Mathematics at Ecole Polytechnique provided by Professor Lucas Gerin. The whole project is implemented through Jupyter Notebook. You can find the [Web Page] and [Source Code] of the project here.

In this project, we focus on the properties and applications of Catalan numbers through numerical algorithms and analytical tools.

- **Chapter 2**, In this chapter we present the most intuitive algorithms for calculating the Catalan number, for example by recursion, iteration, or binomial formula, and their corresponding efficiency analysis.

- **Chapter 3**, In this part, we first compute the generating functions of Catalan numbers and deduce the formula from the generating function. We will also introduce the method with Taylor expansion and compare the efficiency of these approaches. After we will discover Motzkin Numbers and its propositions.

- **Chapter 4**, Since the Catalan numbers increase pretty fast, therefore in this chapter we try to have an analysis using asymptotic analysis as well as algebraic means.

- **Chapter 5**, In the last chapter, we will discover some application in Combinatorial mathematics and some paths algorithmic problems.

Contents

1 Computing Calatan Numbers **2**

2 Generating Function of Catalan Numbers **4**
2.1 Computation of Radius of Convergence for $C(x)$ 5
2.2 Proof of the binomial formula for $c(x)$ 6
2.3 Manipulate Generating Functions with SymPy 7
2.4 Motzkin Numbers 8

3 Dealing with large Catalan numbers **11**
3.1 Catalan and modulos: the Bostan Conjecture 11
3.2 The length of large Catalan numbers 13

4 Some combinatorial interpretations of c_n **16**
4.1 Paths on a triangle 16
4.2 Well-formed parentheses expressions 19
4.3 Binary Trees 20

Initialisation of the Library

```
## loading python libraries
# necessary to display plots inline:
%matplotlib inline

# load the libraries
import matplotlib.pyplot as plt # 2D plotting library
import numpy as np           # package for scientific computing
from pylab import *

from math import *           # package for mathematics (pi, arctan, sqrt, factorial ...)
import sympy as sympy          # package for symbolic computation
```

1 Computing Calatan Numbers

Definition

The Catalan numbers $c_0, c_1, c_2, \ldots$ are defined recursively as follows:

$$c_0 = 1$$
$$c_1 = 1$$
$$c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k} = c_0 c_{n-1} + c_1 c_{n-2} + \cdots + c_{n-1} c_0 \quad (\text{ for } n \geq 2).$$

For example,

$$c_2 = c_0 c_1 + c_1 c_0 = 1 \times 1 + 1 \times 1 = 2,$$
$$c_3 = c_0 c_2 + c_1 c_1 + c_2 c_0 = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5,$$

We use the recursive method and non-recursive method to compute $n - th$ Catalan number which python code. We use a **Python** function to compute it.

```
# Recursive implementation of Catalan numbers
def CatalanRecursive(n):
    if n ==0:
        return 1
    if n ==1:
        return 1
    for i in range(2, n +1):
        Cn = 0
        for j in range(0, i):
            Cn += CatalanRecursive(j) *CatalanRecursive(i -j -1)

    return Cn

# NonRecursive implementation of Catalan numbers
def CatalanNonRecursive(n):
    if n ==0:
        return 1
```

```
    if n ==1:
        return 1
    C = [0] * (n + 1)
    C[0] =1
    C[1] =1
    for i in range(2, n +1):
        C[i] =0
        for j in range(0, i):
            C[i] +=C[j] *C[i -j - 1]

    return C[n]
```

Then we compare the execution times of different functions computing the Catalan numbers (in the range of $1 \leq n \leq 15$ as example). We can plot the graph of the execution time as following. From

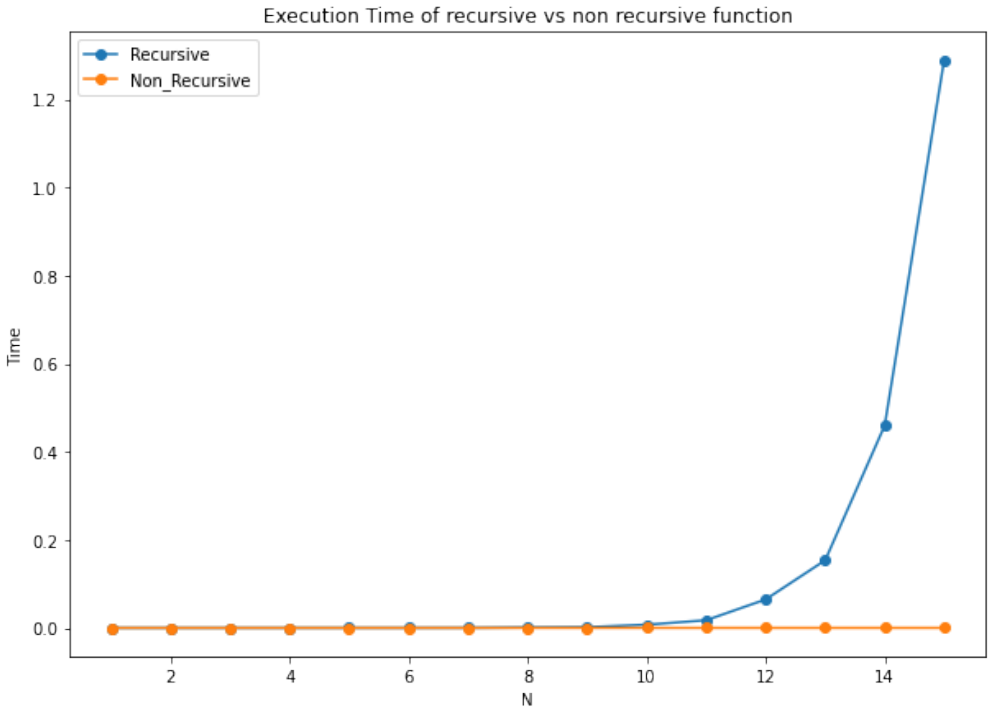


Fig. 1: plot of the execution time of recursive and non-recursive function

the graph that we plot, we can see that the recursive function is much slower than the non-recursive one. From $n = 10$ the recursive function takes much more time than the non-recursive one and **increases exponentially**. This is due to the computation of c_n, the recursive function calls itself too much time with **repeated computation**. However, in the non-recursive function, we use the list to store the values of c_n and we don't need to compute them again.

Propersition 1

It can be proved that for every n

$$c_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

This formula we gonna prove later, then we can deduce from this formula with the python code of following

```
def CatalanClosedForm(n):
    return int((math.factorial(2 *n)) /(math.factorial(n +1) *math.factorial(n)))
```

2 Generating Function of Catalan Numbers

Definition of generating function of Catalan numbers

Let

$$\mathcal{C}(x) = \sum_{n\geq 0} c_n x^n = 1 + x + 2x^2 + 5x^3 + \dots$$

denote the generating function of the Catalan numbers.

We use the ordinary generating function for the Catalan numbers, we have

$$\mathcal{C}(x) = \sum_{n\geq 0} c_n x^n = \sum_{n\geq 0} \sum_{k=0}^{n-1} c_k c_{n-1-k} x^n$$

Since we know $C_0 = 1$, we have

$$\begin{aligned}
\mathcal{C}(x) &= \sum_{n\geq 0} \sum_{k=0}^{n-1} c_k c_{n-1-k} x^n \\
&= 1 + \sum_{n\geq 1} \sum_{k=0}^{n-1} c_k c_{n-1-k} x^n \\
&= 1 + x \sum_{n\geq 0} \sum_{k=0}^{n} c_k c_{n-k} x^n \\
&= 1 + x \left(\sum_{n\geq 0} c_n x^n \right)^2 \\
&= 1 + x\mathcal{C}(x)^2
\end{aligned}$$

Therefore, we got

$$\mathcal{C}(x) = 1 + x\mathcal{C}(x)^2$$

Then we use the recursive formula $(\star)$ to prove that $\mathcal{C}(x)$ is a solution of the following equation of degree two:

$$\mathcal{C}(x) = 1 + x\mathcal{C}(x)^2$$

From the equation $\mathcal{C}(x) = 1 + x\mathcal{C}(x)^2$ we have

$$\mathcal{C}(x) = 1 + x\mathcal{C}(x)^2 \Rightarrow \frac{\mathcal{C}(x) - 1}{x} = \mathcal{C}(x)^2$$

Then we try to prove the **LHS** equal to **RHS**.

For **LHS** we have

$$\begin{aligned}\frac{\mathcal{C}(x)-1}{x} &= \frac{\sum_{n\geq 1} c_n x^n}{x} \\ &= \sum_{n\geq 0} c_{n+1} x^n\end{aligned}$$

For **RHS**, we apply we got

$$\begin{aligned}\mathcal{C}(x)^2 &= (\sum_{n\geq 0} c_n x^n)^2 \\ &= \sum\nolimits_{n\geq 0} (\sum_{k=0}^{n} c_k c_{n-k}) x^n \\ &= \sum\nolimits_{n\geq 0} c_{n+1} x^n \quad \text{(By Definition)}\end{aligned}$$

2.1 Computation of Radius of Convergence for $C(x)$

```
# Computation with sympy of the equation
x = Symbol('x')
c = Symbol('c')
solution =solve(c -1 - x * c**2, c)
```

We use **sympy** to solve the equation $\mathcal{C}(x) = 1 + x\mathcal{C}(x)^2$ and we get

$$\mathcal{C}_1(x) = \frac{1-\sqrt{1-4x}}{2x} \quad \text{and} \quad \mathcal{C}_2(x) = \frac{1+\sqrt{1-4x}}{2x}$$

From the two possibilities, the $\mathcal{C}_1(x)$ must be chosen because only the choice of the first gives

$$C_0 = \lim_{x\to 0} \mathcal{C}_1(x) = 1$$

Therefore we have $\mathcal{C}(x) = \frac{1-\sqrt{1-4x}}{2x}$

Then we compute the radius of convergence of $\mathcal{C}(x)$. For the definition of **[radius of convergence]** from Wiki page. We use the expression of $\mathcal{C}(x)$ in power series that

$$\mathcal{C}(x) = \frac{1-\sqrt{1-4x}}{2x} = \sum_{n\geq 0} c_n x^n$$

Base on **[D'Alembert's test]**, for radius of convergence R we compute $\lim_{n\to\infty} \left|\frac{c_{n+1}}{c_n}\right| = \rho$ for $\sum_{n\geq 0} c_n x^n$.

$$\begin{aligned}
\rho &= \lim_{n\to\infty} |\frac{c_{n+1}}{c_n}| \\
&= \lim_{n\to\infty} \left| \frac{\frac{1}{n+2}\binom{2n+2}{n+2}}{\frac{1}{n+1}\binom{2n}{n}} \right| \\
&= \lim_{n\to\infty} \left| \frac{4n+4}{n+2} \right| \\
&= \lim_{n\to\infty} \left| 4 - \frac{4}{n+2} \right| \\
&= 4
\end{aligned}$$

Since $\rho \neq 0$, then we have the radius of convergence $R = \frac{1}{\rho} = \frac{1}{4}$

Then we can compare the computing efficiency of a different approach

1. Recursive Method
2. Non-recursive Method
3. Compute with formula $c_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$

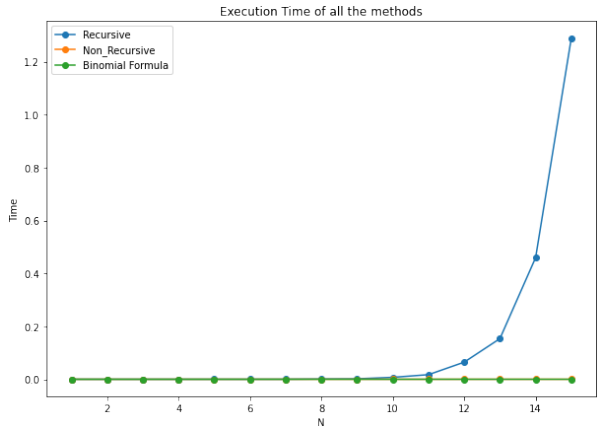


Fig. 2: Execution Time of all the methods

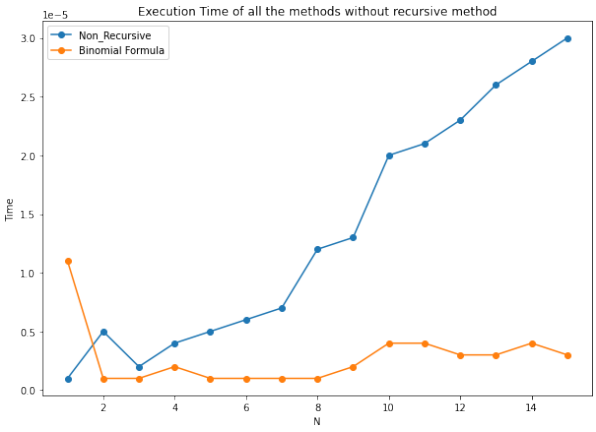


Fig. 3: Execution Time of Non-recursive VS computation with formula

From the two graph, we can see that the formula of $c_n = \frac{1}{n+1}\binom{2n}{n}$ has the highest computation efficiency. However, the non-recursive method has more execution time than the method above, compare with the recursive method the complexity is still much better since the execution time is at the level of $10^{-5} sec$.

2.2 Proof of the binomial formula for $c(x)$

We can prove the expression of

$$c_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

with $\mathcal{C}(x) = \frac{1-\sqrt{1-4x}}{2x} = \sum_{n\geq 0} c_n x^n$

For the power series we have

$$\sqrt{1+y} = \sum_{n=0}^{\infty} \binom{\frac{1}{2}}{n} y^n = \sum_{n=0}^{\infty} \frac{(-1)^{n+1}}{4^n(2n-1)} \binom{2n}{n} y^n$$

We denote that $y = -4x$ we got

$$\sqrt{1-4x} = 1 - \sum_{n=1}^{\infty} \frac{1}{2n-1} \binom{2n}{n} x^n$$

Therefore we got

$$\mathcal{C}(x) = \frac{1-\sqrt{1-4x}}{2x} = \sum_{n=1}^{\infty} \frac{1}{2(2n-1)} \binom{2n}{n} x^{n-1} = \sum_{n\geq 0} c_n x^n$$

We can deduce that

$$c_n = \frac{1}{2(2n+1)} \binom{2n+2}{n+1} = \frac{1}{n+1} \binom{2n}{n}$$

2.3 Manipulate Generating Functions with SymPy

Example for an Taylor Expansion

We work out with the example of

$$f(x) = \frac{1}{1-2x} = 1 + 2x + 4x^2 + 8x^3 + 16x^4 + \dots$$

We first introduce variable x and function f as follows:

```
x=var('x')
f=(1/(1-2*x))

print('f = '+str(f))
print('series expansion of f at 0 and of order 10 is: '+str(f.series(x,0,10)))
display(f.series(x,0,10))
```

We have the Taylor expansion of $f(x)$ in degree 10 that

$$1 + 2x + 4x^2 + 8x^3 + 16x^4 + 32x^5 + 64x^6 + 128x^7 + 256x^8 + 512x^9 + O\left(x^{10}\right)$$

We can use this idea to compute the Catalan numbers using $\mathcal{C}(x)$.

```
# We compute C(x) with Taylor Expansion
x = var('x')
f = (1 - sqrt(1 -4 * x)) / (2 * x)

def CatalanTaylorExpansion(n):
    return f.series(x, 0, n).coeff(x**(n -1))
```

```
lis =[CatalanTaylorExpansion(i) for i in range(11)]
lis[1] =1
print("The first ten Catalan numbers with Taylor Expansion are ", lis[1:])
```

We have the output that: The first ten Catalan numbers with Taylor Expansion are [1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862] Then we compare the complexity of methods:

1. Recursive Method
2. Non-recursive Method
3. Compute with formula $c_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$
4. Compute with Taylor Expansion

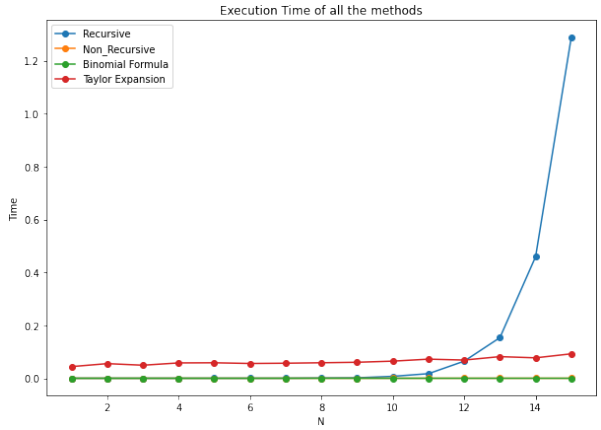


Fig. 4: Execution Time of all the methods

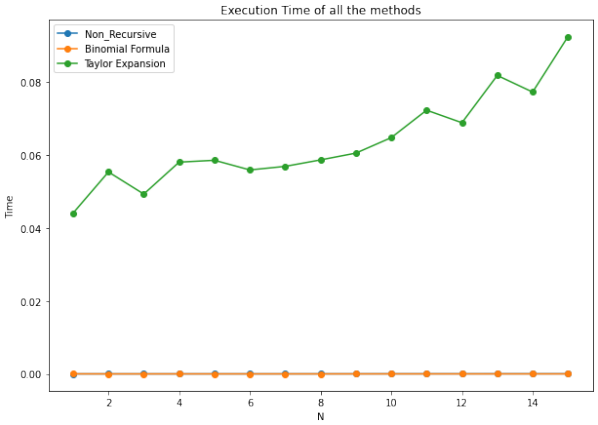


Fig. 5: Execution Time of methods without Recursive

From the two graph we can see that the Taylor Expansion method is much faster than the recursive method. However, the non-recursive method and method of computation with formula have better efficiency than the Taylor Expansion method.

2.4 Motzkin Numbers

Definiton of Motzkin numbers

The **Motzkin numbers** $m_0, m_1, m_2, \ldots$ are similar to Catalan numbers and defined by $m_0 = m_1 = 1$ and for every $n \geq 2$

$$m_n = m_{n-1} + \sum_{k=0}^{n-2} m_k m_{n-2-k}.$$

We first try to simplify the motzkin numbers recurrence relation. We have

$$\mathcal{M}(x) = \sum_{n\geq 0} m_n x^n = \sum_{n\geq 0}(m_{n-1} + \sum_{k=0}^{n-2} m_k m_{n-2-k})x^n$$

Since we have $\mathcal{M}_0 = \mathcal{M}_1 = 1$, therefore we have

$$\begin{aligned}
\mathcal{M}(x) &= \sum_{n\geq 0}(m_{n-1} + \sum_{k=0}^{n-2} m_k m_{n-2-k})x^n \\
&= x\sum_{n\geq 0} m_{n-1}x^{n-1} + \sum_{n\geq 0}\sum_{k=0}^{n-2} m_k m_{n-2-k}x^n \\
&= x\mathcal{M}(x) + 1 + \sum_{n\geq 2}\sum_{k=0}^{n-2} m_k m_{n-2-k}x^n \\
&= x\mathcal{M}(x) + 1 + x^2\sum_{n\geq 0}\sum_{k=0}^{n} m_k m_{n-k}x^n \\
&= x\mathcal{M}(x) + 1 + x^2(\sum_{n\geq 0} m_n x^n)^2 \\
&= x\mathcal{M}(x) + 1 + x^2\mathcal{M}(x)
\end{aligned}$$

Therefore, we know that $\mathcal{M}(x) = \sum_{n\geq 0} m_n x^n$ is a root for $x^2M^2 + (x-1)M + 1 = 0$.

Then we use **SymPy** to find the generating function $\mathcal{M}(x) = \sum_{n\geq 0} m_n x^n$.

```
x = symbols('x')
M = symbols('M')
SeriesM =solve(x**2 *M**2 +x * M - M + 1, M)
print(SeriesM)
display(SeriesM[0])
display(SeriesM[1])
```

We know that $\mathcal{M}(x) = \sum_{n\geq 0} m_n x^n$ is a root for $x^2M^2 + (x-1)M + 1$. We use **SymPy** to solve the equation we have

$$\mathcal{M}_1(x) = \frac{1 - x - \sqrt{1-2x-3x^2}}{2x^2} \quad \mathcal{M}_2(x) = \frac{1 - x + \sqrt{1-2x-3x^2}}{2x^2}$$

Since we know $\mathcal{M}(0) = 1$, therefore the only possible solution is

$$\mathcal{M}(x) = \frac{1 - x - \sqrt{1-2x-3x^2}}{2x^2}$$

```
# Implementation of the formula of Motzkin Numbers
def MotzkinFormula(n):
    x = symbols('x')
    m= (1-x-sqrt(1-2*x-3*x**2))/(2*x**2)
    m_truncated =m.series(x,0,n+1)
    return m_truncated.coeff(x**n)
```

An integral representation of Motzkin numbers is given by

$$m_n = \frac{2}{\pi}\int_0^{\pi} \sin(x)^2(2\cos(x)+1)^n dx.$$

They have the asymptotic behaviour

$$m_n \sim \frac{1}{2\sqrt{\pi}} \left(\frac{3}{n}\right)^{3/2} 3^n, n \to \infty$$

Therefore, they have the same radius of convergence. We apply a similar method to the Catalan number. Base on **[D'Alembert's test]**, for radius of convergence R we compute $\lim_{n\to\infty} \left|\frac{m_{n+1}}{m_n}\right| = \rho$ for $\sum_{n\geq 0} m_n x^n$

$$\begin{aligned}
\rho &= \lim_{n\to\infty} |\frac{m_{n+1}}{m_n}| \\
&= \lim_{n\to\infty} \left|\frac{(\frac{3}{n+1})^{\frac{3}{2}} 3^{n+1}}{(\frac{3}{n})^{\frac{3}{2}} 3^n}\right| \\
&= \lim_{n\to\infty} \left|3(\frac{n}{n+1})^{3/2}\right| \\
&= 3
\end{aligned}$$

Since $\rho \neq 0$, then we have the radius of convergence $R = \frac{1}{\rho} = \frac{1}{3}$.

Or we can illustrate in this way, from the definition of the radius of convergence, we need to find an R such that when $\|x\| < R$ the series converges. Then we observe that the radius shows when $1 - 2R - 3R^2 = 0$. Then we got $R = \frac{1}{3}$ or $r = -1$. Since when $x < 1$, the function diverges, we can also get $R = \frac{1}{3}$

Then we can say the radius of convergence of $\mathcal{C}$ is smaller than $\mathcal{M}$. Therefore the sequence of c_n is growing faster than m_n. We can also show this in the graph.

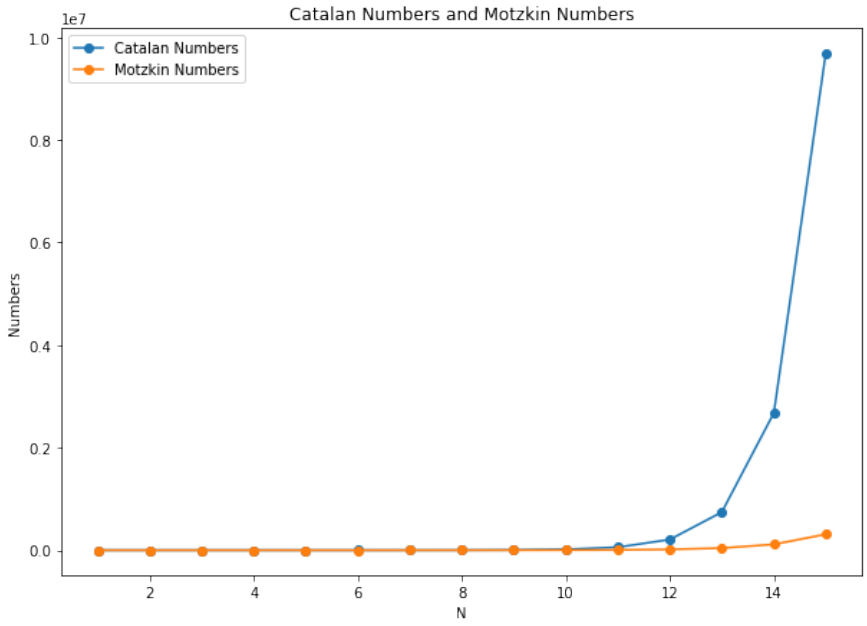


Fig. 6: Increasing Speed of Catalan Numbers and Motzkin Numbers

3 Dealing with large Catalan numbers

3.1 Catalan and modulos: the Bostan Conjecture

Definiton of Bostan Conjecture

Alin Bostan (computer scientist at INRIA and Ecole Polytechnique) conjectured a few years ago see this link p.26 that in basis 10 the last digit of c_n is never 3. So far this is still an open problem.

Check the conjecture for $1 \leq n \leq 100$. The output should look like

```
Catalan 1 mod 10 is 1: the Conjecture is True
Catalan 2 mod 10 is 2: the Conjecture is True
Catalan 3 mod 10 is 5: the Conjecture is True
Catalan 4 mod 10 is 4: the Conjecture is True
...
```

Since the number is relatively small, we can apply the naive algorithms that

```
# We test the conjecture for 1 to 100
for n in range(1, 101):
    if CatalanClosedForm(n) % 10 !=3:
        print('Catalan', n, 'mode 10 is',
              CatalanClosedForm(n) % 10, ':the Conjecture is', True)
    else:
        print('the Conjecture is', False)
```

From the output that the conjecture is true for $1 \leq n \leq 100$. However, in order to verify the conjecture for very large values, we need to apply the new algorithms.

Since our previous function is not that efficient, we need to apply other algorithms. We use the property that

$$(a \cdot b) \bmod p = ((a \bmod p) \cdot (b \bmod p)) \bmod p$$

This is easy to prove, we assume that $k_1 = (a \bmod p)$ and $k_2 = (b \bmod p))$. Therefore, we have $a = mq + k_1$ and $b = nq + k_2$ where $m, n \in \mathbb{Z}$. We have

$$\begin{aligned}(a \cdot b) \bmod p &= (mnq^2 + (mk_2 + nk_1)q + k_1k_2) \bmod p \\ &= (k_1k_2) \bmod p \\ &= ((a \bmod p) \cdot (b \bmod p)) \bmod p\end{aligned}$$

Then we can use this property to write the recursive function as above.

```
def precalc(MOD, num):
    catalan =[0] *num
    catalan[0], catalan[1] =1, 1
    for i in range(2, num):
        total =0
        for j in range(1, i+1):
            left =catalan[j-1] % MOD
```

```
            right =catalan[i-j] % MOD
            total =(total +(left *right) % MOD) % MOD
        catalan[i] =total
    return catalan

if __name__ ==’__main__’:
    catalan =precalc(10, 8000)
    for i in range(7000, 7101):
        if catalan[i] !=3:
            print(’Catalan’, i, ’mode 10 is’, catalan[i], ’:the Conjecture is’,
                True)
        else:
            print(’the Conjecture is’, False)
```

After all the computations, we find that the conjecture is true for $0 \leq n \leq 7101$. Also, we can test the frequency for the last digits from 0 to 9. We tested the case for $n = 1000$, $n = 5000$, and $n = 10000$. We have the result as follows

```
--------The test case with n =1000 --------
The frequency of the last digit 1 is 0.002
The frequency of the last digit 2 is 0.056
The frequency of the last digit 5 is 0.004
The frequency of the last digit 4 is 0.073
The frequency of the last digit 9 is 0.003
The frequency of the last digit 0 is 0.781
The frequency of the last digit 6 is 0.03
The frequency of the last digit 8 is 0.05
The frequency of the last digit 7 is 0.001
--------The test case with n =5000 --------
The frequency of the last digit 1 is 0.0004
The frequency of the last digit 2 is 0.0292
The frequency of the last digit 5 is 0.0014
The frequency of the last digit 4 is 0.0414
The frequency of the last digit 9 is 0.0006
The frequency of the last digit 0 is 0.8692
The frequency of the last digit 6 is 0.0224
The frequency of the last digit 8 is 0.0352
The frequency of the last digit 7 is 0.0002
--------The test case with n =10000 --------
The frequency of the last digit 1 is 0.0002
The frequency of the last digit 2 is 0.0244
The frequency of the last digit 5 is 0.0008
The frequency of the last digit 4 is 0.0319
The frequency of the last digit 9 is 0.0003
The frequency of the last digit 0 is 0.9021
The frequency of the last digit 6 is 0.0162
The frequency of the last digit 8 is 0.024
The frequency of the last digit 7 is 0.0001
```

We test the case for $n = 1000$, $n = 5000$, and $n = 10000$. From the bar chart and the output of our test, we may deduce that in basis 10 the last digit of c_n is never 3. When n increases, we can find the following

- The probability for the last digits of Catalan numbers is 0 getting increases.
- The probability for the last digits of Catalan numbers is even number getting decreasing.
- The probability of the last digits of Catalan numbers being an odd number (3 not included) is really small, which means the last digits of Catalan numbers are rarely odd when n is a large number.

For the visualization graph, we can plot the frequency as a bar chat

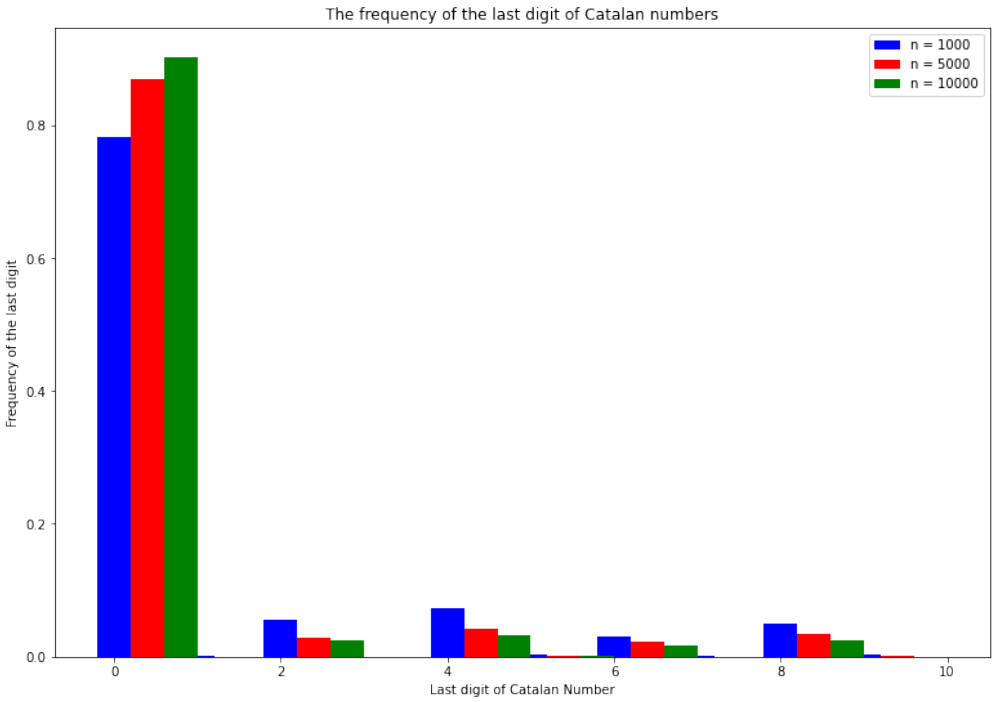


Fig. 7: Frequency of the last digit of Catalan numbers

3.2 The length of large Catalan numbers

Problem of the length for Catalan number

It can be proved (this is beyond the level of Bachelor 2, a possible reference is p.384 in Ph.Flajolet, R.Sedgewick, *Analytic Combinatorics*) that for every n we have

$$\frac{4^n}{\sqrt{\pi n^3}}\left(1-\frac{9}{8n}\right) \leq c_n \leq \frac{4^n}{\sqrt{\pi n^3}}$$

which yields very good approximations when n is large. We will use this approximation to estimate the *length* (*i.e.* the number of digits) of c_n when n is a power of ten. Consider the following table which records the *length* of c_{10}, c_{100}, c_{1000},...

c_{10^n}	Number of digits of c_{10^n}
c_{10}	5
c_{100}	57
c_{10^3}	598
c_{10^4}	6015
c_{10^5}	60199

Propersition 2

We have a formula that gives the number of digits of a given integer. For any positive integer n, we assume the digits of n are i. We have the relation that

$$10^{i-1} \leq n < 10^i$$

Then we take the $\log_{10}$ for both side we have

$$i - 1 \leq \log(n) < i$$

Then we can prove that the digits of any given number n is

$$i = \lfloor \log(n) \rfloor + 1$$

From the equation from above we have $i = \lfloor \log(n) \rfloor$ and $\frac{4^n}{\sqrt{\pi n^3}}(1 - \frac{9}{8n}) \leq c_n \leq \frac{4^n}{\sqrt{\pi n^3}}$. Then we take the $\log_{10}$ for both side we have

$$\lfloor \log(\frac{4^n}{\sqrt{\pi n^3}}(1 - \frac{9}{8n})) \rfloor + 1 \leq i = \lfloor \log(c_n) \rfloor \leq \lfloor \log(\frac{4^n}{\sqrt{\pi n^3}}) \rfloor + 1$$

Since n is a number of power 10, then we denote $n = 10^k$ where $k \geq 1$ and k is an integer. Then we have

$$\begin{aligned}
\log(\frac{4^n}{\sqrt{\pi n^3}}(1 - \frac{9}{8n})) &= \log(\frac{4^n}{\sqrt{\pi n^3}}) + \log(1 - \frac{9}{8n}) \\
&= \log 4^n - (\frac{1}{2}\log(\pi) + \frac{3}{2}\log(n)) + \log(1 - \frac{9}{8n}) \\
&= 10^k \log 4 - (\frac{1}{2}\log(\pi) + \frac{3k}{2}) + \log(1 - \frac{9}{8n})
\end{aligned}$$

Since $k \geq 1$ and $n = 10^k$, then we know that n goes really fast, therefore $\log(1 - \frac{9}{8n})$ tend close to 0 and we can neglect it. Then we have

$$i = \lfloor \log(\frac{4^n}{\sqrt{\pi n^3}}(1 - \frac{9}{8n})) \rfloor + 1 = \lfloor 10^k \log 4 - (\frac{1}{2}\log(\pi) + \frac{3k}{2}) \rfloor + 1$$

Therefore, we can use the code below to complete the table.

```
# implement the formula above
def num_digits(n):
    ub = (10**n) *log10(4) -(1 / 2) * log10(pi) -(3 *n / 2) * log10(10)
    return math.floor(ub) +1
```

We get the output that

```
The number of digits of Catalan, 10 **1 is 5
The number of digits of Catalan, 10 **2 is 57
The number of digits of Catalan, 10 **3 is 598
The number of digits of Catalan, 10 **4 is 6015
The number of digits of Catalan, 10 **5 is 60199
The number of digits of Catalan, 10 **6 is 602051
The number of digits of Catalan, 10 **7 is 6020590
The number of digits of Catalan, 10 **8 is 60205987
The number of digits of Catalan, 10 **9 is 602059978
The number of digits of Catalan, 10 **10 is 6020599899
The number of digits of Catalan, 10 **11 is 60205999117
The number of digits of Catalan, 10 **12 is 602059991310
The number of digits of Catalan, 10 **13 is 6020599913260
```

We can see from the result that Cat_{10} grows really fast. The number of digits of Cat_{10} is 5 and the number of digits of $Cat_{10}$is 602051. Also. for larger and larger n's the right column always begins with the same digits (60205...). From the formula we have

$$i = \lfloor 10^k \log 4 - (\frac{1}{2}\log(\pi) + \frac{3k}{2}) \rfloor + 1$$

Since we want to prove that the right column always begins with the same digits (60205...), then we can ignore the smaller part of i, which means $(\frac{1}{2}\log(\pi) + \frac{3k}{2})$ can be ignored since there are quite small compared to $10^k \log 4$. Then compute $10^k \log 4$ with **Python** code and we have the value of $10^k \log(4)$ for $k \in [1, 20]$. Since $\log(4) = 0.6020599913279624$, so we consider the marginal case, when $k = 6$, $10^6 \log(4) = 602059.9913279624$, and $\frac{1}{2}\log(\pi) + \frac{3k}{2} < 9$, therefore it would not influnce the second digits of i. This also applies to the value of i with bigger k. Then we prove the theorem.

4 Some combinatorial interpretations of c_n

4.1 Paths on a triangle

Information of the Path on a triangle model

Let $\mathcal{T} \subset \mathbb{N}^2$ denote the infinite "triangle"

$$\mathcal{T} = \{(k, n), \quad 0 \leq k \leq n\}$$

(see the figure below).
For $(k, n) \in \mathcal{T}$ we denote by $P_{k,n}$ the number of paths such that:

- the path starts at $(0, 0)$ ends at (n, k) and entirely lies inside $\mathcal{T}$
- the paths only takes unit steps in the North and East directions.

For example this figure shows that $P_{2,3} = 5$:

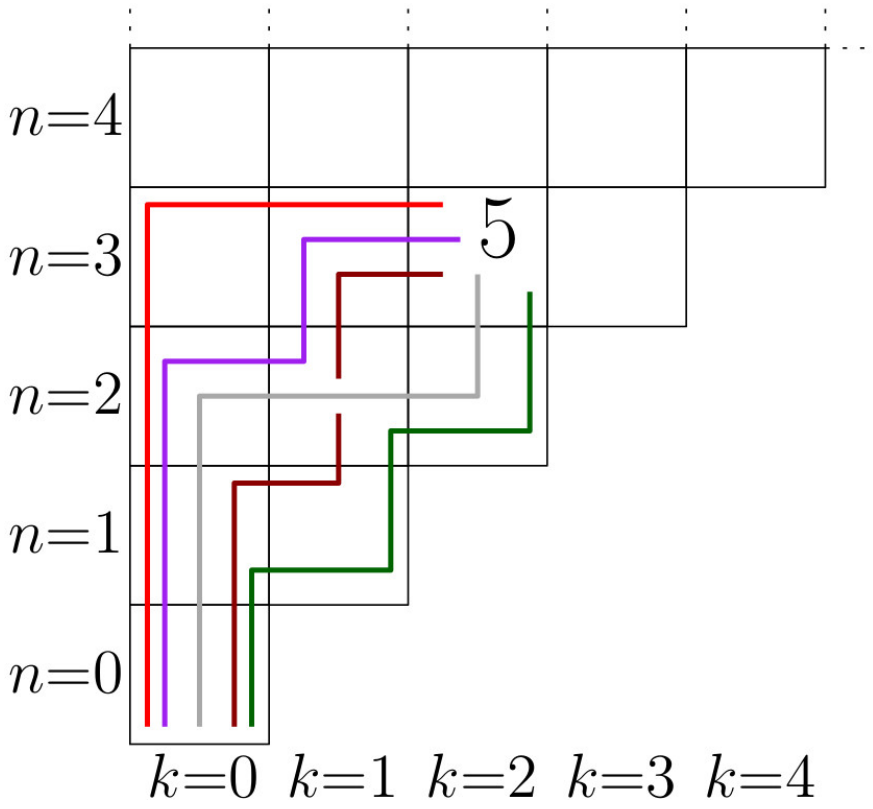


Fig. 8: Figure of the Infinite Triangle

We used the python code to analyze this model, **Paths(K, N)** which returns a table (or a matrix) of all the values of $P_{k,n}$ for $k \leq K, n \leq N$.

```
# implement of the function Path
# we use the recursive method by implement the auxiliary function Path
def Path(k, n):
```

```
    result =[['']]
    n_up =[[0]]
    n_right =[[0]]
    if n ==0:
        return []
    if k ==0:
        return ['up ' *n]
    for i in range(n +k +1):
        result.append([])
        n_up.append([])
        n_right.append([])
        for j in range(len(result[i])):
            r = result[i][j]
            if n_up[i][j] <n:
                result[i +1].append(r +' up ')
                n_up[i +1].append(n_up[i][j] +1)
                n_right[i +1].append(n_right[i][j])
            if n_right[i][j] <k and n_right[i][j] <n_up[i][j]:
                result[i +1].append(r +'right ')
                n_up[i +1].append(n_up[i][j])
                n_right[i +1].append(n_right[i][j] +1)
    return result[-2]


def Paths(K, N):
    result =np.zeros((K +1, N +1))
    for k in range(K +1):
        for n in range(N +1):
            result[k, n] =len(Path(k, n))

    return result

print("--------- Another Test Sample ----------")
print(Paths(5, 6))

---------Output of Another Test Sample ----------
[[  0.   1.   1.   1.   1.   1.   1.]
 [  0.   1.   2.   3.   4.   5.   6.]
 [  0.   0.   2.   5.   9.  14.  20.]
 [  0.   0.   0.   5.  14.  28.  48.]
 [  0.   0.   0.   0.  14.  42.  90.]
 [  0.   0.   0.   0.   0.  42. 132.]]
```

We denote the sequence $u_k = P_{k,k+1}$. From the matrix we can find that u_k is same with the sequence of the Calatan Number, which means $P_{k,k+1} = c_k = \sum_{j=0}^{k-1} = c_k c_{n-1-k}$. Now lets prove it.

First we try to discuss in a more simple model. For example, I try to compute $P_{3,4} = 14$. For each points $(k, k+1)$ where $1 \leq k \leq 3$, we consider those points as step points, which means for example, we consider $(2, 3)$ as an intermediate point, we have for all the possible paths to $(3, 4)$ go through $(2, 3)$, is the multiple of $P_{2,3} \cdot P_{2,1} = c_0 c_2$. For each step point, the paths go through the step points we can consider the process of two parts, from $(0, 0)$ to the step points and from step

points to $(k, k+1)$.

Then we can expand this model into more general cases. we try to compute $P_{k,k+1}$, the for all step points $(m, m+1)$ where $0 \leq m \leq k-1$. For all the possible paths to $P_{k,k+1}$, we know it go through at least one step point, since all the paths start from $(0,0)$ and has to go through $(0,1)$ cause we have to go up at the beginning. Therefore we can consider all the paths into to step. First from $(0,0)$ to the step point, which we have $P_{m,m+1}$ and from the step point to $(k, k+1)$, which is equivalent to from $(0,1)$ to $(k-m-1, k-m)$, we have $P_{k-m-1,k-m}$ possibilities. Then number of paths to $(k, k+1)$ is the multiplication of $P_{m,m+1}$ and $P_{k-m-1,k-m}$. So we have the sum for all step points that

$$P_{k,k+1} = \sum_{m=0}^{k-1} P_{k-m-1,k-m} P_{m,m+1}$$

Since $P_{0,1} = 1$ and $P_{1,2} = 2$. And $c_k = \sum_{j=0}^{k-1} = c_k c_{n-1-k}$ by the recursive formula of the Catalan number. Therefore we prove that

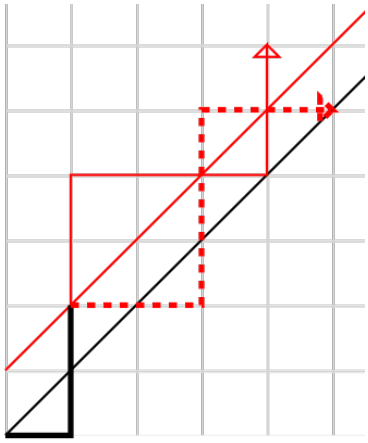
$$P_{k,k+1} = c_k = \sum_{j=0}^{k-1} = c_k c_{n-1-k}$$

Also we have $P_{k,k}$ is also Catalan number, which is easy to explain since $P_{k,k} = P_{k,k+1}$, since from (k,k) to $(k,k+1)$ there only on way which is go on step on direction East, so we have

$$P_{k,k} = P_{k,k+1} = c_k = \sum_{j=0}^{k-1} = c_k c_{n-1-k}$$

So the paths to diagonal positions are also Catalan numbers.

We can also use another method to prove **Path(K, K)** is Catalan Number. The method is from [Wiki page] of Catalan number. We count the number of paths which start and end on the diagonal of a $n \times n$ grid. All such paths have n right and n up steps. Since we have $2n$ steps in total, therefore all the possibilities are binomial number $\binom{2n}{n}$. However, we are restrict by the triangle. As the



graph shown above, the path below the diagonal line is not valid. Therefore, we use make the invalid portion of the path flipped like the graph shown. Since we have $2n$ steps in total and now we have $n+1$ up then we got $n-1$ right steps. Due to this reason, we reach at $(n-1, n+1)$ in the end. Because every monotonic path in the $(n-1) \times (n+1)$ grid meets the higher diagonal, and because the reflection process is reversible, the reflection is therefore a bijection between bad paths in the original grid. Therefore, for the bad paths we got

$$\begin{pmatrix} n-1+n+1 \\ n-1 \end{pmatrix} = \begin{pmatrix} 2n \\ n-1 \end{pmatrix} = \begin{pmatrix} 2n \\ n+1 \end{pmatrix}$$

And the valid paths of the Catalan paths are

$$C_n = \begin{pmatrix} 2n \\ n \end{pmatrix} - \begin{pmatrix} 2n \\ n+1 \end{pmatrix} = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix}$$

4.2 Well-formed parentheses expressions

Information of the well-formed parentheses expressions

It can be shown that c_n counts the number of expressions containing n pairs of parentheses which are **correctly matched**. For the first values we obtain

$n = 1:$ ()
$n = 2:$ (()) ()()
$n = 3:$ ((())) (())() ()(()) ()()() (()())

We write a recursive function "Parentheses(n)" which returns the list of all well-formed parentheses expressions with n pairs of parentheses.

We can use the model above, "Path to Triangle" in order to solve this problem. We can consider "(" as a step in the North direction and ")" as a step in the East direction. Then in this way, We transfer the problem "Parentheses(n)" to the problem "Paths(n, n)". Therefore, we can apply a similar method to solve this problem.

```
# Implement a similar method with the function Path
def Parentheses(n):
    result =[['(']]
    n_open =[[1]]
    n_close =[[0]]
    for i in range(2 *n):
        result.append([])
        n_open.append([])
        n_close.append([])
        for j in range(len(result[i])):
            r = result[i][j]
            if n_open[i][j] <n:
                result[i +1].append(r +'(')
                n_open[i +1].append(n_open[i][j] +1)
                n_close[i +1].append(n_close[i][j])
            if n_close[i][j] <n_open[i][j]:
                result[i +1].append(r +')')
```

```
            n_open[i +1].append(n_open[i][j])
            n_close[i +1].append(n_close[i][j] +1)
    return result[-2]
```

As we mentioned above, the Well-formed parentheses expressions 'Parentheses(k)' are exactly the same in the Paths on triangle problem from $(0, 0)$ to (k, k), 'Paths(k,k)'.

We can consider (as a step in the North direction and (as a step in the East direction. For every well-formed parentheses expression, we start from (, and the condition for well-formed parentheses is at any point, the number of (is larger than). For example, ()((())), we can see that at any index of this expression, the number of (before the index is larger than (. These conditions correspond to the condition in the Paths on triangle model that the first step we go up, and the steps in the North direction are always larger than the steps in the East direction otherwise we are out of the triangle board. From $(0, 0)$ to (k, k), we have the same number of steps in the North and East direction which is the same that we have the same amount of (and). Therefore we prove that **Parentheses(k)** is counted by Catalan numbers.

Or we explain in this way. We have $p(0) = 1$ as there is only one way to arrange no parentheses and $p(1) = 1$.

For $n \geq 2$, we can consider the well-formed parentheses expression as two parts, **(part1)part2** where part 1 and part 2 are split by a pair of rackets. Then we assume part 1 has k pairs of parentheses and part 2 has $n - 1 - k$ pairs of parentheses. Since $k \in [0, n - 1]$. Therefore we have the expression

$$p_n = \sum_{k=0}^{n-1} p_k p_{n-1-k}$$

This agrees with the formula of the Catalan number.

4.3 Binary Trees

A **binary tree** is a tree in which every internal node (in grey in the above pictures) has exactly two children. Leaves (in green) have no children. The **size** of a binary tree is its number of internal nodes. There is one binary tree of size 1, and two binary trees of size 2, five binary trees of size 3:

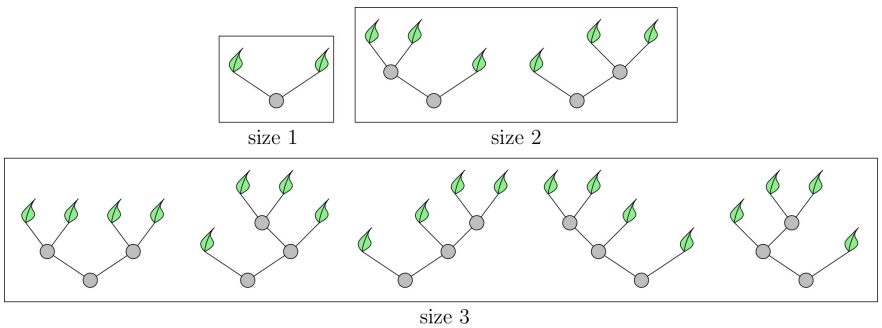


Fig. 9: Figure of the Infinite Triangle

Let t_n be the number of binary trees of size n, by convention we put $t_0 = 1$ (this corresponds to a leaf without any internal node).

For the Catalan numbers, we have the definition that

$$\begin{aligned} c_0 &= 1 \\ c_1 &= 1 \\ c_n &= \sum_{k=0}^{n-1} c_k c_{n-1-k} = c_0 c_{n-1} + c_1 c_{n-2} + \cdots + c_{n-1} c_0 \qquad (\text{ for } n \geq 2). \end{aligned}$$

Which can also be rewritten as:

$$\begin{aligned} c_0 &= 1 \\ c_1 &= 1 \\ c_{n+1} &= \sum_{k=0}^{n} c_k c_{n-k} \end{aligned}$$

For $t_1 = 1$ we have zero parent so there only one case. Then we consider more general cases for $n \geq 2$. We imagine a tree t with $n+1$ nodes has one root with two subtrees as children t_1 and t_2. Since the root of t is a parent node, t_1 and t_2 must have n parent nodes together (i.e the sum of nodes for t_1 and t_2 is n). A subtree t_1 or t_2 can be empty. Therefore the ways of organize t_1 which k nodes is $T_k T_{n-k}$. Then we have

$$\sum_{i=0}^{n} T_i T_{n-i}$$

Since $i \in [0, n]$. So we finish the prove that the n-th Catalan number.