A journey from binomial coefficients to fractals and mathematical art

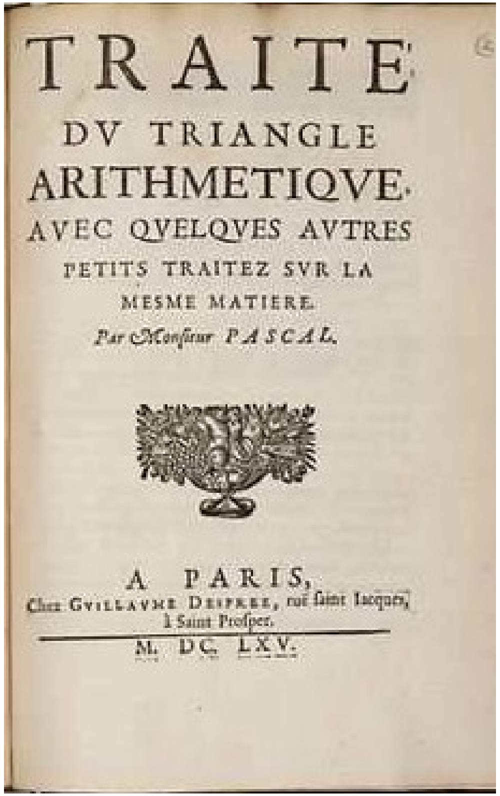
Robert CABANE
Mathematician (retired)

About Blaise Pascal (1623-1662)

French mathematician, physicist, inventor, philosopher, writer, and theologian.

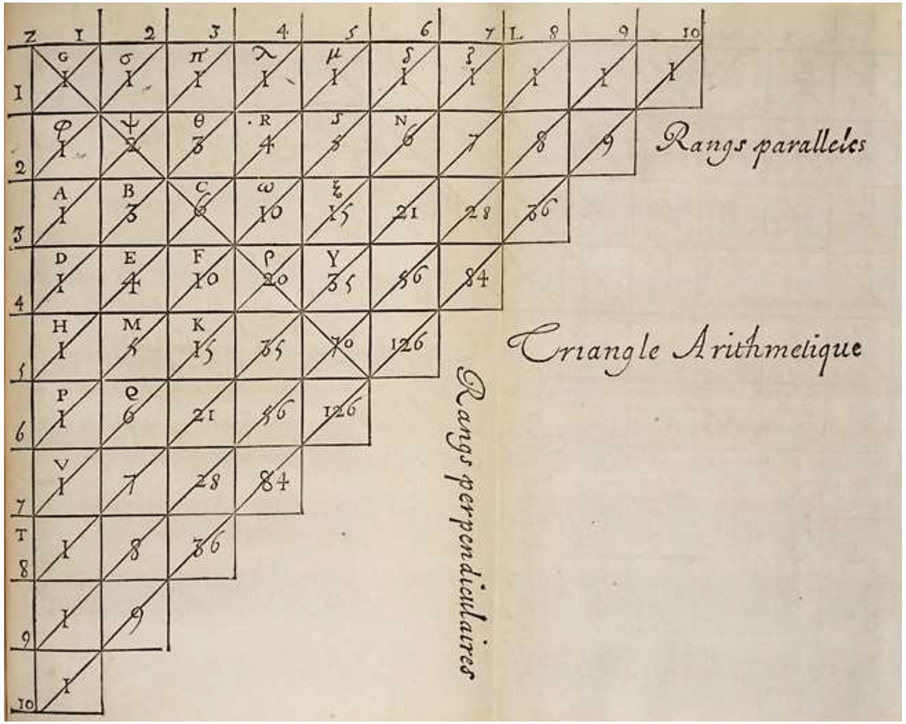
Picture : Wikimedia

Le traité du triangle arithmétique


TRAITE
DV TRIANGLE
ARITHMETIQVE.
AVEC QVELQVES AVTRES
PETITS TRAITEZ SVR LA
MESME MATIERE.
Par Monsieur PASCAL.

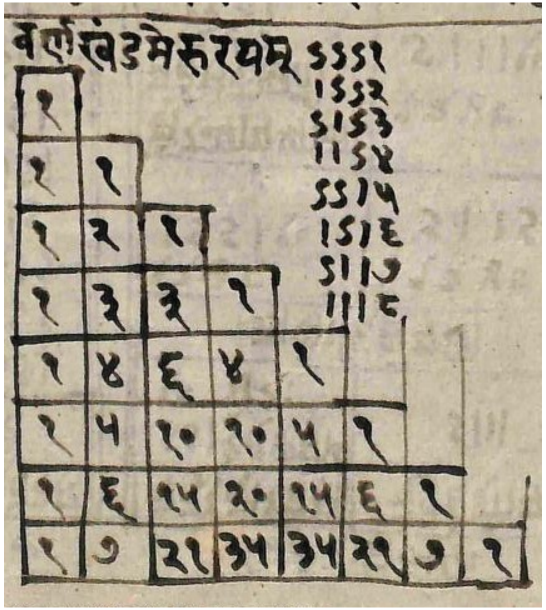
A PARIS,
Chez GVILLAVME DESPREZ, ruë saint Iacques,
à Saint Prosper.
M. DC. LXV.

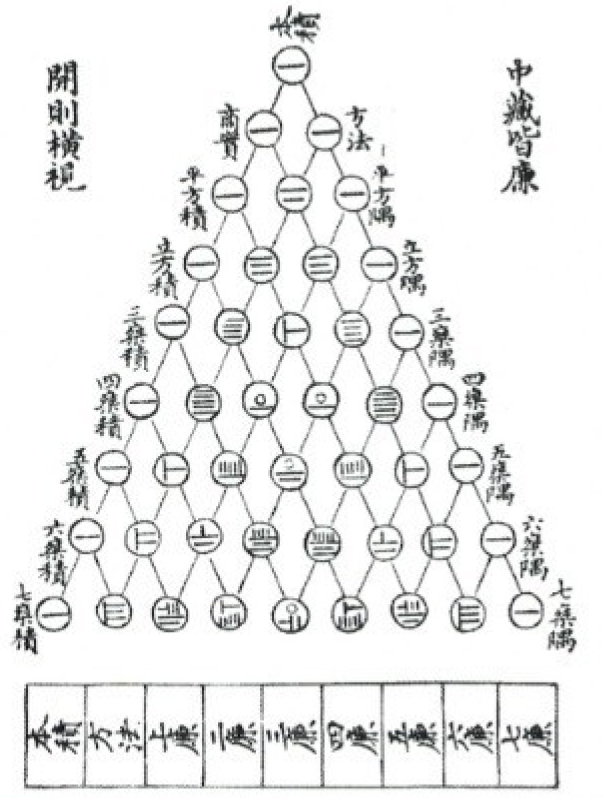
What we now call “Pascal’s triangle” was published by him in 1655 as “triangle arithmétique”.



Not so new

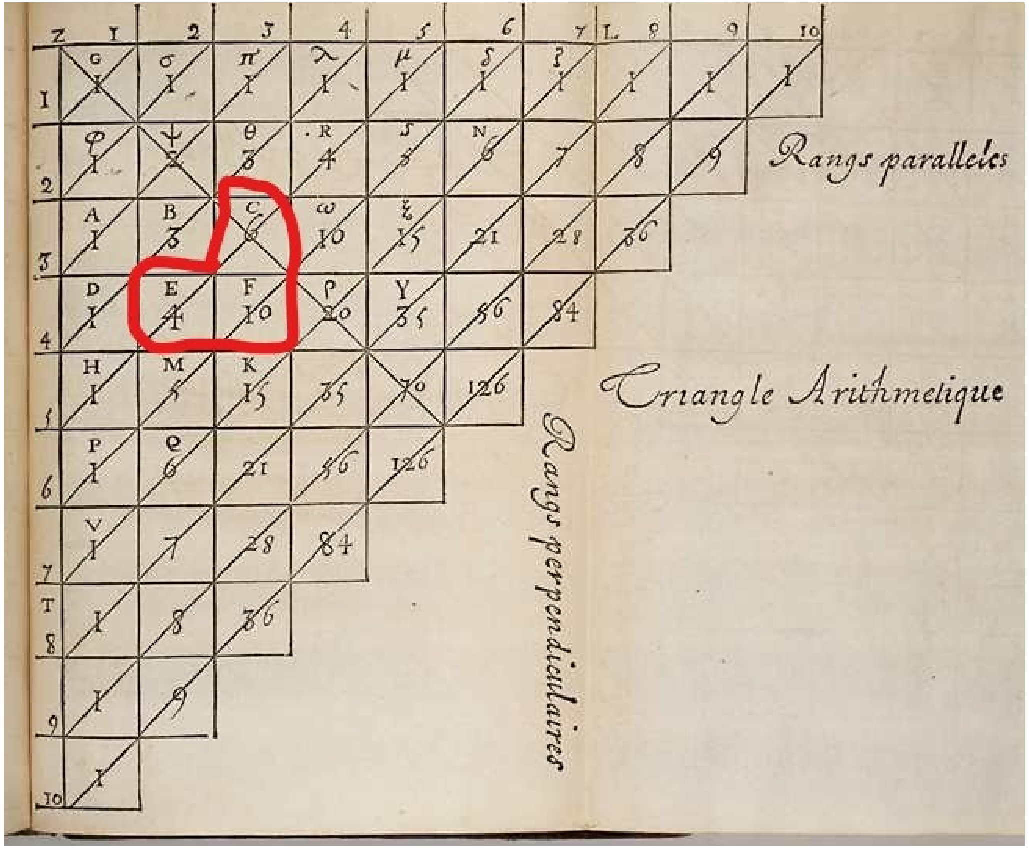
- The binomial coefficients were already known to Chinese mathematicians (although Pascal wasn't aware).
- Yang Hui 1238-1298 (published in a book of Zhu Shijie, dated 1303)
- They also appeared in India, many centuries before …





Pictures : Wikimedia

The triangle rule



A simple rule

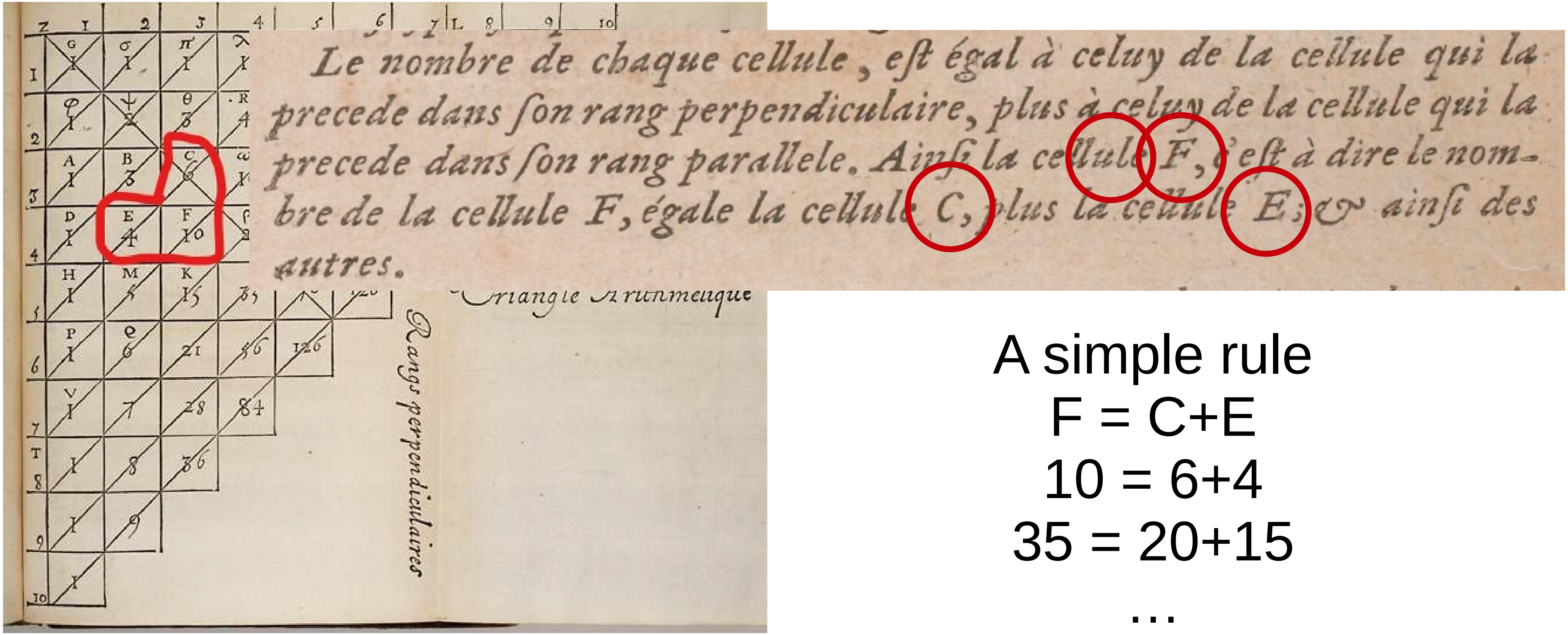
F = C+E

10 = 6+4

35 = 20+15

…

The triangle rule



A simple rule

F = C+E

10 = 6+4

35 = 20+15

…

In modern words

Pascal's design was later modified in order to better take in account the binomial theorem… "sliding" columns a bit.

k=2 (column)

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

← Row 5 (diagonal in Pascal design)

In modern words

Pascal's design was later modified in order to better take in account the binomial theorem.

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

k=2 ↓

← n=5

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Row
(diagonal in Pascal's design)

Column

And the binomial theorem

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

$$(1+x)^2 = 1+2x+x^2$$

$$(1+x)^3 = 1+3x+3x^2+x^3$$

$$(1+x)^4 = 1+4x+6x^2+4x^3+x^4$$

$$(1+x)^5 = 1+5x+10x^2+10x^3+5x^4+x^5$$

Binomial coefficients, in 4 ways

How can we compute the binomial coefficients ?

Good news ! Python computes easily with large integers ☺.

Binomial coefficients, in 4 ways

How can we compute the binomial coefficients ?

Good news ! Python computes easily with large integers ☺.

- Using the triangle rule: [1] $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
- Using a direct formula: [2] $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$

Binomial coefficients, in 4 ways

How can we compute the binomial coefficients ?

Good news ! Python computes easily with large integers ☺.

- Using the triangle rule: [1] $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
- Using a direct formula: [2] $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$
- Using a recursive scheme: [3] $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ with $\binom{m}{0} = \binom{m}{m} = 1$

Binomial coefficients, in 4 ways

How can we compute the binomial coefficients ?

Good news ! Python computes easily with large integers ☺.

- Using the triangle rule: [1] $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
- Using a direct formula: [2] $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$
- Using a recursive scheme: [3] $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ with $\binom{m}{0} = \binom{m}{m} = 1$
- Using factorials: [4] $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ with $n! = n(n-1)\cdots 2\cdot 1$ (and $0! = 1$)

Go on along your own way!

- Now you can program if you want. Start Python on your Nspire CX-II software or hand-held (or on the 84 CE Python edition, it works also), and try to create such a function :

```
def binom(n,k):
  ...
  return ...
```

Hint : the quotient of the division of `m` by `j` (taken as *integers*) should be coded as **m//j** (m/j being a float).

- Your code should remain *short* (4-5 lines, no more).
- You can test your function asking for binom(500,214) (a bunch of digits, finishing by 06000).

Some possible Python codes

[2] $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1\cdots(k-1)k}$

[3] $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$

[4] $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Some possible Python codes

[2] $$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1\cdots(k-1)k}$$

```
def binom2r(n,k):
    if k==0: return 1
    return binom2r(n,k-1)*(n-k+1)//k

def binom2i(n,k):
    X=1
    for i in range(1,k+1): X=(X*(n-i+1))//i
    return X
```

Here we have a recursive function (e.g. a function calling itself). Use with care.

… and here an iterative function doing the same computations.

Some possible Python codes

[2] $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1\cdots(k-1)k}$

```
def binom2r(n,k):
    if k==0: return 1
    return binom2r(n,k-1)*(n-k+1)//k

def binom2i(n,k):
    X=1
    for i in range(1,k+1): X=(X*(n-i+1))//i
    return X
```

```
def binom3r(n,k):
    if k==0: return 1
    return binom3r(n-1,k-1)*n//k

def binom3i(n,k):
    X=1
    for j in range(1,k+1): X=(X*(n-k+j))//j
    return X
```

[3] $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$

Some possible Python codes

[2] $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1\cdots(k-1)k}$

```
def binom2r(n,k):
  if k==0: return 1
  return binom2r(n,k-1)*(n-k+1)//k

def binom2i(n,k):
  X=1
  for i in range(1,k+1): X=(X*(n-i+1))//i
  return X
```

```
def binom3r(n,k):
  if k==0: return 1
  return binom3r(n-1,k-1)*n//k

def binom3i(n,k):
  X=1
  for j in range(1,k+1): X=(X*(n-k+j))//j
  return X
```

[3] $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$

It's better to avoid recursivity (Python has its limits…)

```
def facto(n):
# Factorial of an integer
  p=1
  for k in range(1,n+1): p=p*k
  return p

def binom4(n,k):
# Binomial coefficient, based upon factorials
  return (facto(n)//facto(k))//facto(n-k)
```

[4] $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Code : `binomial.tns` `binomial.8xv`

The worse possible code

- Back to scheme [1] (triangle rule): what about recursivity ?

```
def recbin(n,k):
  if k==0 or n==k: return 1
  return recbin(n-1,k)+recbin(n-1,k-1) # triangle rule
```

- It works, indeed. But … let's try it !

The worse possible code

- Back to scheme [1] (triangle rule): what about recursivity ?

```
def recbin(n,k):
  if k==0 or n==k: return 1
  return recbin(n-1,k)+recbin(n-1,k-1) # triangle rule
```

- It works, indeed. But … let’s try it !
- The `recbin(25,9)` call lasts 30 seconds on my CX-II hand-held.
- Why ?

The worse possible code

- Back to scheme [1] (triangle rule): what about recursivity ?

```
def recbin(n,k):
  if k==0 or n==k: return 1
  return recbin(n-1,k)+recbin(n-1,k-1) # triangle rule
```

- It works, indeed. But … let's try it !
- The `recbin(25,9)` call lasts 30 seconds on my CX-II hand-held.
- Why ? Just doubling the calls at each step … $2^{25}>3.10^{7}$ calls !
- Other codes fail with "maximum recursion depth exceeded"

One coefficient vs. one row

- Another approach to the binomial coefficients : compute whole rows of the triangle, filling a list with the help of the triangle rule [1].
- A single list is here enough if we accept an “overloading” process, e.g. starting with L=[1,2,1,0,0] it’s possible to modify terms of L like this, processing from right to left :

L[3] = L[3]+L[2] # gives 1

L[2] = L[2]+L[1] # gives 3

L[1] = L[1]+L[0] # gives 3

L[0] unchanged (still 1) ⇒ L=[1,3,3,1,0]

One coefficient vs. one row

- Another approach to the binomial coefficients : compute whole rows of the triangle, filling a list with the help of the triangle rule [1].
- A single list is here enough if we accept an “overloading” process, e.g. starting with L=[1,2,1,0,0] it’s possible to modify terms of L like this, processing from right to left :

L[3] = L[3]+L[2] # gives 1

L[2] = L[2]+L[1] # gives 3

L[1] = L[1]+L[0] # gives 3

L[0] unchanged (still 1) ⇒ L=[1,3,3,1,0]

```
def line(n): # computes Pascal's triangle line
  n=n+1 ; L=[0]*n ; L[0]=1 #initializations
  for i in range(n):
    for j in range(i,0,-1): # RTL
      L[j]=L[j-1]+L[j]   # overwriting
  return L
```

Code : `binomial.tns`

One coefficient vs. one row

- Another approach to the binomial coefficients : compute whole rows of the triangle, filling a list with the help of the triangle rule [1].
- A single list is here enough if we accept an “overloading” process, e.g. starting with L=[1,2,1,0,0] it’s possible to modify terms of L like this, processing from right to left :

  L[3] = L[3]+L[2] # gives 1
  L[2] = L[2]+L[1] # gives 3
  L[1] = L[1]+L[0] # gives 3
  L[0] unchanged (still 1) ⇒ L=[1,3,3,1,0]

- Caution : processing from left to right doesn’t work.

```
def line(n): # computes Pascal's triangle line
  n=n+1 ; L=[0]*n ; L[0]=1 #initializations
  for i in range(n):
    for j in range(i,0,-1): # RTL
      L[j]=L[j-1]+L[j]   # overwriting
  return L
```

```
>>>line(7)
[1, 7, 21, 35, 35, 21, 7, 1]
>>>line(8)
[1, 8, 28, 56, 70, 56, 28, 8, 1]
```

Display the triangle (1)

- Now we can show Pascal's triangle.
- The code is very similar, appending a copy of the computed "row" L to a list (of lists) P.
- Caution : if you just code P.append(L) you get a mess ...

```
def line(n): # computes a Pascal's triangle line
  n=n+1 ; L=[0]*n ; L[0]=1 #initializations
  for i in range(n):
    for j in range(i,0,-1): # RTL
      L[j]=L[j-1]+L[j]    # overwriting
  return L

def triangle(n): # prints Pascal's triangle
    n=n+1 ; P=[]
    L=[0]*n ; L[0]=1 #initializations
    for i in range(n):
        for j in range(i,0,-1): # RTL
            L[j]=L[j-1]+L[j]    # overwriting
        # list(L) creates a new list from L
        P.append(list(L))
    return P
```

Display the triangle (1)

- Now we can show Pascal's triangle.
- The code is very similar, appending a <u>copy</u> of the computed "row" `L` to a list (of lists) `P`.
- We print here the successive lists contained in the output list (e.g. `P`).

```
>>>for s in triangle(11): print(s)
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 3, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 4, 6, 4, 1, 0, 0, 0, 0, 0, 0, 0]
[1, 5, 10, 10, 5, 1, 0, 0, 0, 0, 0, 0]
[1, 6, 15, 20, 15, 6, 1, 0, 0, 0, 0, 0]
[1, 7, 21, 35, 35, 21, 7, 1, 0, 0, 0, 0]
[1, 8, 28, 56, 70, 56, 28, 8, 1, 0, 0, 0]
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1, 0, 0]
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1, 0]
[1, 11, 55, 165, 330, 462, 462, 330, 165, 55, 11, 1]
```

Display the triangle (2)

- Colouring the numbers according to their parity, some patterns seem to appear.

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

Display the triangle (2)

- Colouring the numbers according to their parity, some patterns seem to appear.
- Changing these figures to **1** = odd, **0** = even and later to pixels (1 = coloured pixel, 0 = white pixel) will give us many patterns to explore.

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

1																
1	1															
1	0	1														
1	1	1	1													
1	0	0	0	1												
1	1	0	0	1	1											
1	0	1	0	1	0	1										
1	1	1	1	1	1	1	1									
1	0	0	0	0	0	0	0	1								
1	1	0	0	0	0	0	0	1	1							
1	0	1	0	0	0	0	0	1	0	1						
1	1	1	1	0	0	0	0	1	1	1	1					
1	0	0	0	1	0	0	0	1	0	0	0	1				
1	1	0	0	1	1	0	0	1	1	0	0	1	1			
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Display the triangle – graphically (1)

- Stephen Wolfram (author of Mathematica) published a paper about this idea in 1984 in his paper "*Geometry of binomial coefficients*" in the Amer. Math. Monthly.

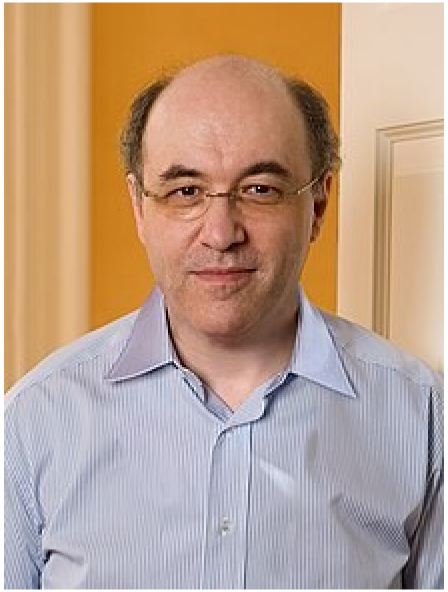


Photo : Wikipedia

Display the triangle – graphically (2)

- Stephen Wolfram (author of Mathematica) published a paper about this idea in 1984 in his paper “*Geometry of binomial coefficients*” in the Amer. Math. Monthly.

- Let’s program this in Python with the Nspire CX-II. The list L receives successive lines of Pascal’s triangle, as before, and points are plotted in red when the coefficient is odd.

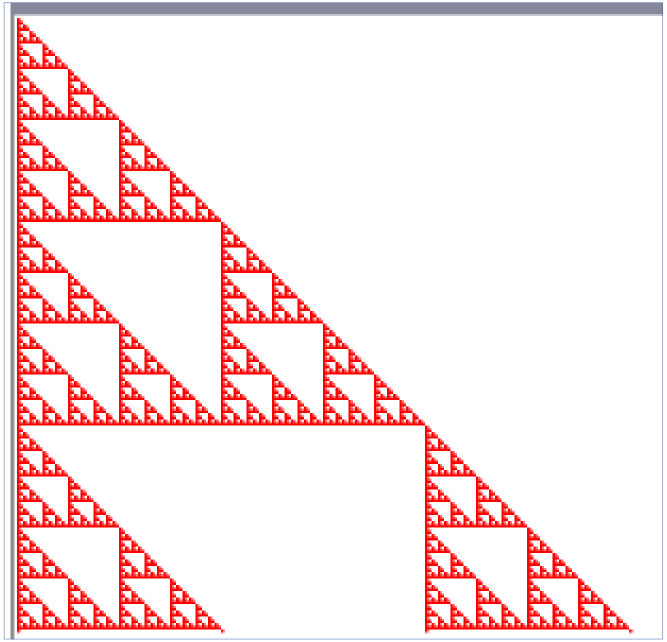
```
from ti_draw import *

def pt(i,j): # plot a single point
  plot_xy(1+j,1+i,7)
def t(p): # draw the triangle
  clear() ; set_color(255,0,0)
  n=p+1 ; L=[0]*n ; L[0]=1
  for i in range(n):
    for j in range(i,-1,-1):
      if j>0: L[j]=L[j-1]+L[j]
      if L[j]%2==1: pt(i,j)
```

Code : `pascal.tns / pascal.8xv`

Display the triangle – graphically (3)

- Stephen Wolfram (author of Mathematica) published a paper about this idea in 1984 in the "*Geometry of binomial coefficients*" in the Amer. Math. Monthly.
- Let's program this in Python with the Nspire CX-II. The list L receives successive lines of Pascal's triangle, as before, and points are plotted in red when the coefficient is odd.
- The resulting figure is here ⇒⇒⇒



Display the triangle – symmetrically

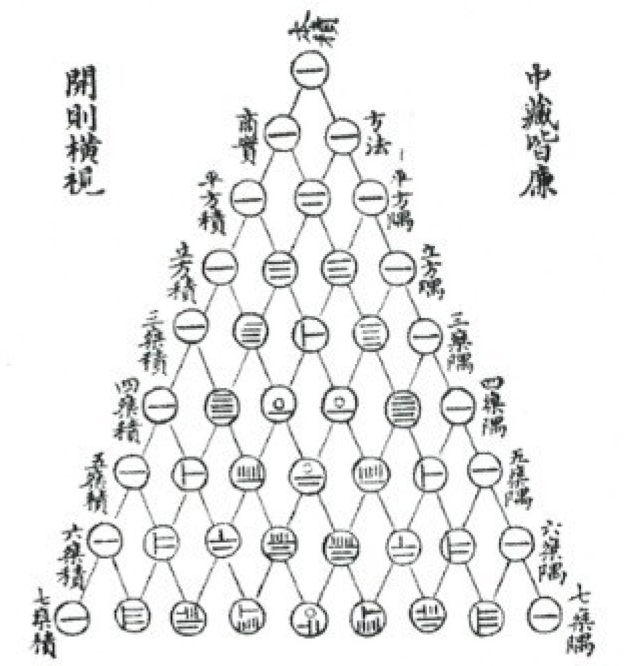
- Among the many patterns of Pascal’s triangle, there is a symmetry, due to the formula shown here on the right.

$$\binom{n}{k} = \binom{n}{n-k}$$

Display the triangle – symmetrically

- Among the many patterns of Pascal's triangle, there is a symmetry, due to the formula shown here on the right.
- In order to better "see" this symmetry, just dispose the triangle in Yang Hui's way :

$$\binom{n}{k} = \binom{n}{n-k}$$

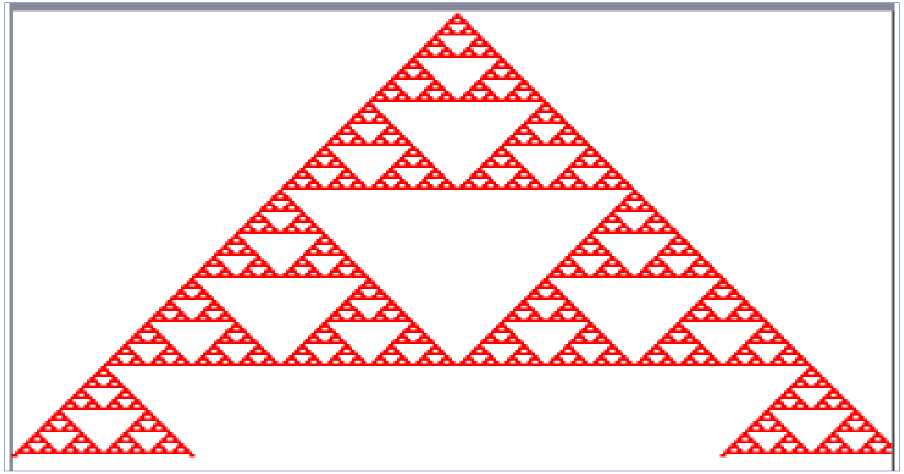


Display the triangle – symmetrically

- Among the many patterns of Pascal’s triangle, there is a symmetry, due to the formula shown here on the right.
- In order to better “see” this symmetry, just dispose the triangle in Yang Hui’s way.
- The algorithm is very similar : just change the `pt` function.

$$\binom{n}{k} = \binom{n}{n-k}$$

```
def pt2(i,j): # plot a point, better
    plot_xy(160-i+2*j,1+i,7)
```



Display the triangle – graphically (4)

This “triangles in triangle” design was first imagined by Wacław Sierpiński, polish mathematician (1882-1969).

Photo : Wikipedia

Fig. 1.

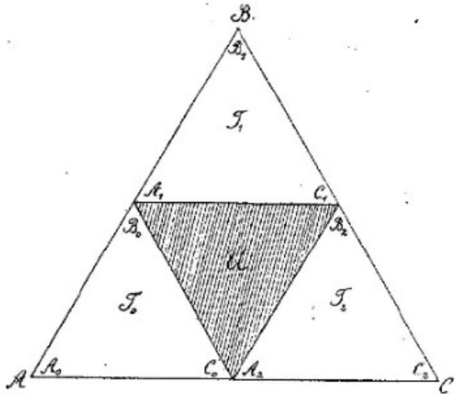
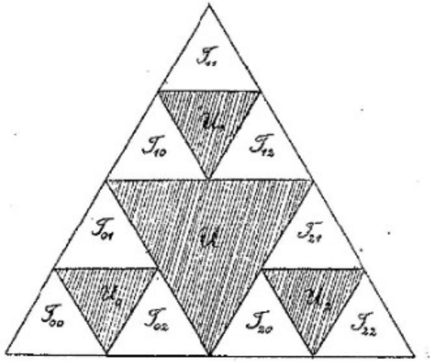


Fig. 2.



Display the triangle – graphically (5)

This “triangles in triangle” design was first imagined by Wacław Sierpiński, polish mathematician (1882-1969). He published an article in 1915 about the now so-called “Sierpiński gasket”, one of the first examples of a *fractal* curve (the “fractal” word appeared much later).

Photo : Wikipedia

Fig. 1.

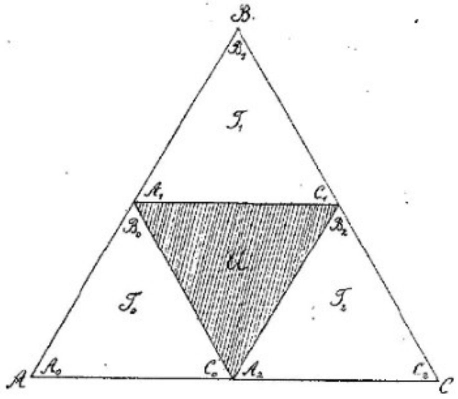
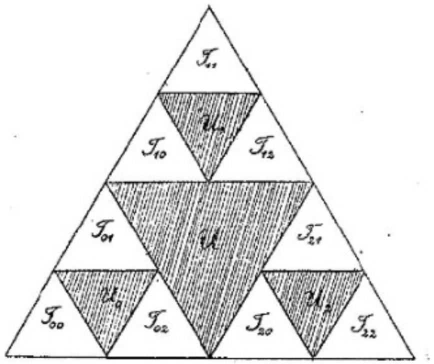


Fig. 2.



Why this ?

The self-similarity of the triangle taken modulo a prime number (here, 2) was discovered by the french mathematician Édouard Lucas (in 1878). Lucas was a math teacher whose research didn't receive due support at his time, and his main article (excerpt below) isn't easy to read.

Photo : Wikipedia

On a donc, en général, pour p premier,

$$C_m^n \equiv C_{m_1}^{n_1} \times C_\mu^\nu \quad (\text{mod. } p),$$

m_1 et n_1 désignant les entiers de $\frac{m}{p}$ et de $\frac{n}{p}$, et μ et ν les résidus de m et de n suivant le module p.

An insight into the Lucas theorem (1)

Lemma. If $2^s > c > 0$, then $\binom{2^s}{c}$ is even. Equivalently, the only odd coefficients of the 2^s row are the extreme ones.

Consequence. In the $2^s - 1$ row of the triangle, all coefficients are odd.

Proof. Recall the formula $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ (for $k>0$), or $k\binom{n}{k} = n\binom{n-1}{k-1}$. So we have $c\binom{2^s}{c} = 2^s\binom{2^s-1}{c-1}$ (because $c>0$). The RHS has at least s times 2 in factor, while in the LHS the factor c has at most $s-1$ times 2 in factor since $c<2^s$. Thus, the binomial $\binom{2^s}{c}$ has to be even.

An insight into the Lucas theorem (2)

Lemma. If $2^s > c > 0$, then $\binom{2^s}{c}$ is even. Equivalently, the only odd coefficients of the 2^s row are the extreme ones.

Consequence. In the $2^s - 1$ row of the triangle, all coefficients are odd.

1																
1	1															
1	0	1														
1	1	1	1													
1	0	0	0	1												
1	1	0	0	1	1											
1	0	1	0	1	0	1										
1	1	1	1	1	1	1	1									
1	0	0	0	0	0	0	0	1								
1	1	0	0	0	0	0	0	1	1							
1	0	1	0	0	0	0	0	1	0	1						
1	1	1	1	0	0	0	0	1	1	1	1					
1	0	0	0	1	0	0	0	1	0	0	0	1				
1	1	0	0	1	1	0	0	1	1	0	0	1	1			
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

An insight into the Lucas theorem (2)

Lemma. If $2^s > c > 0$, then $\binom{2^s}{c}$ is even. Equivalently, the only odd coefficients of the 2^s row are the extreme ones.

Consequence. In the $2^s - 1$ row of the triangle, all coefficients are odd.

We can see this here, looking at the rows # 3, 7, 15 (beware : the triangle starts with a row #0, consisting of a single 1).
For instance, row #3 consists of four ones.

1																
1	1															
1	0	1														
1	1	1	1													
1	0	0	0	1												
1	1	0	0	1	1											
1	0	1	0	1	0	1										
1	1	1	1	1	1	1	1									
1	0	0	0	0	0	0	0	1								
1	1	0	0	0	0	0	0	1	1							
1	0	1	0	0	0	0	0	1	0	1						
1	1	1	1	0	0	0	0	1	1	1	1					
1	0	0	0	1	0	0	0	1	0	0	0	1				
1	1	0	0	1	1	0	0	1	1	0	0	1	1			
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

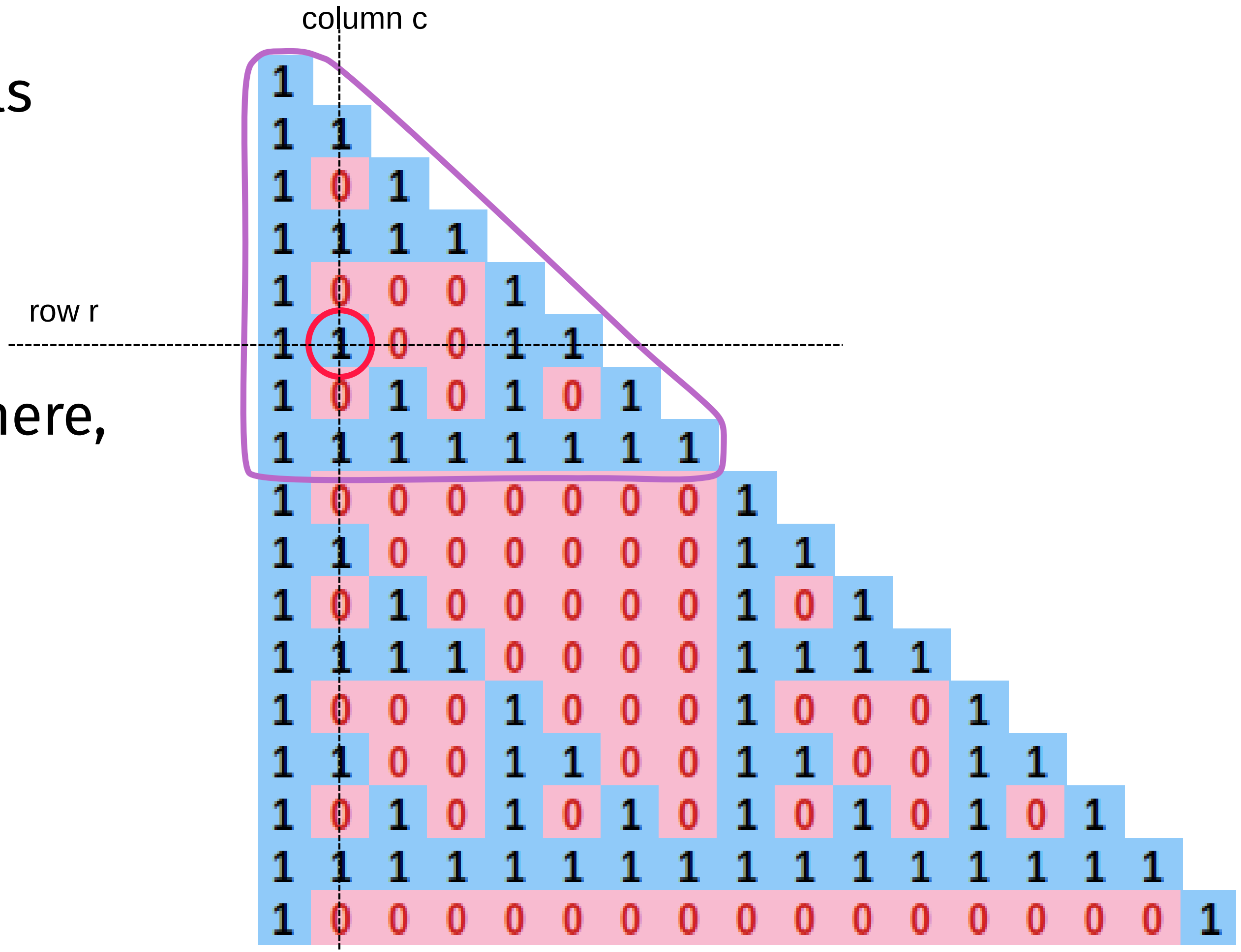
An insight into the Lucas theorem (3)

Lucas again. The binomials

$$\binom{r}{c}, \binom{r+2^s}{c} \text{ and } \binom{r+2^s}{c+2^s}$$

have the same parity.

We can observe this fact here, with $r=5$, $c=2$ and $2^s=8$.



An insight into the Lucas theorem (3)

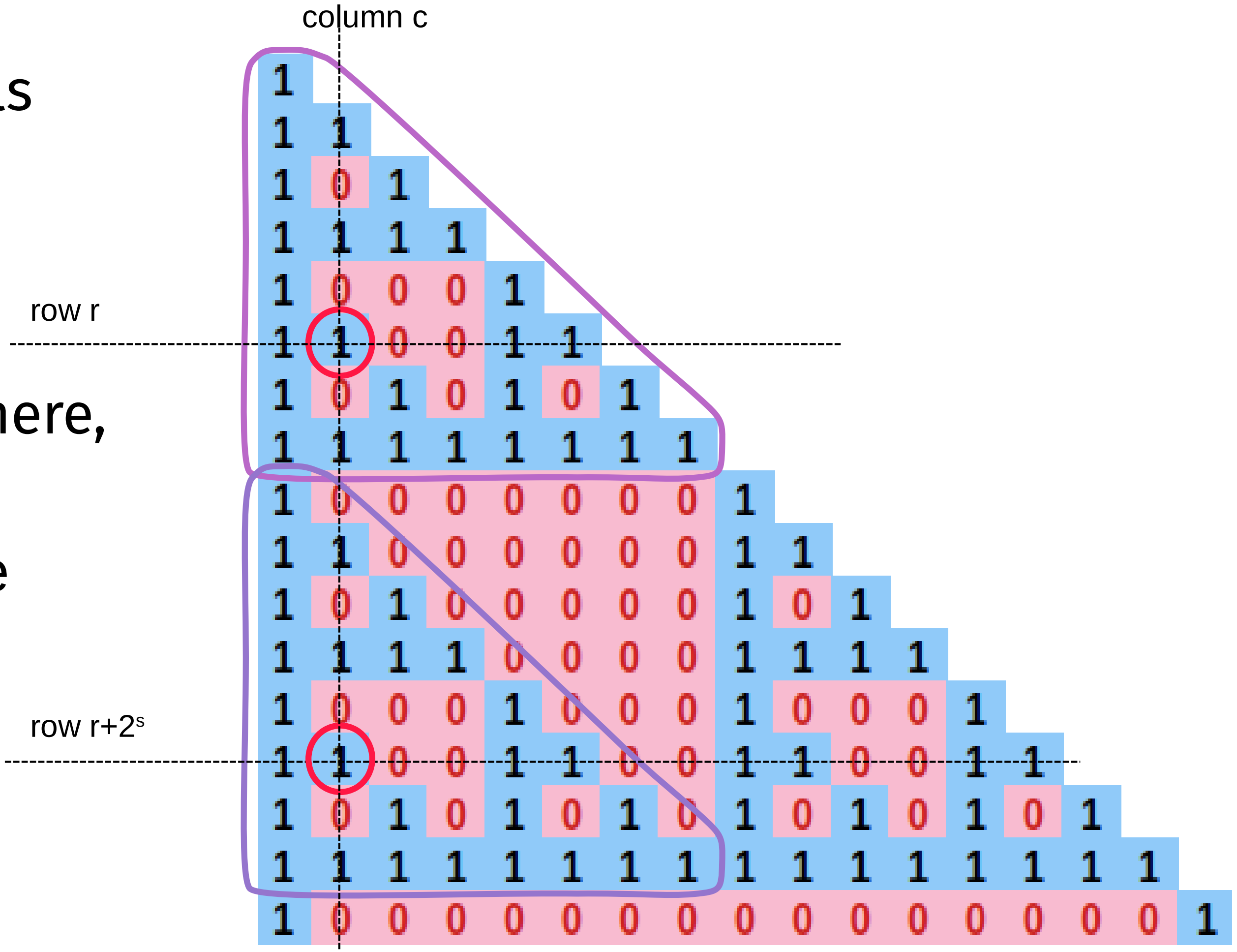
Lucas again. The binomials

$$\binom{r}{c}, \binom{r+2^s}{c} \text{ and } \binom{r+2^s}{c+2^s}$$

have the same parity.

We can observe this fact here, with $r=5$, $c=2$ and $2^s=8$.

The upper "triangle", made of 8 rows, gets a copy below ...



An insight into the Lucas theorem (3)

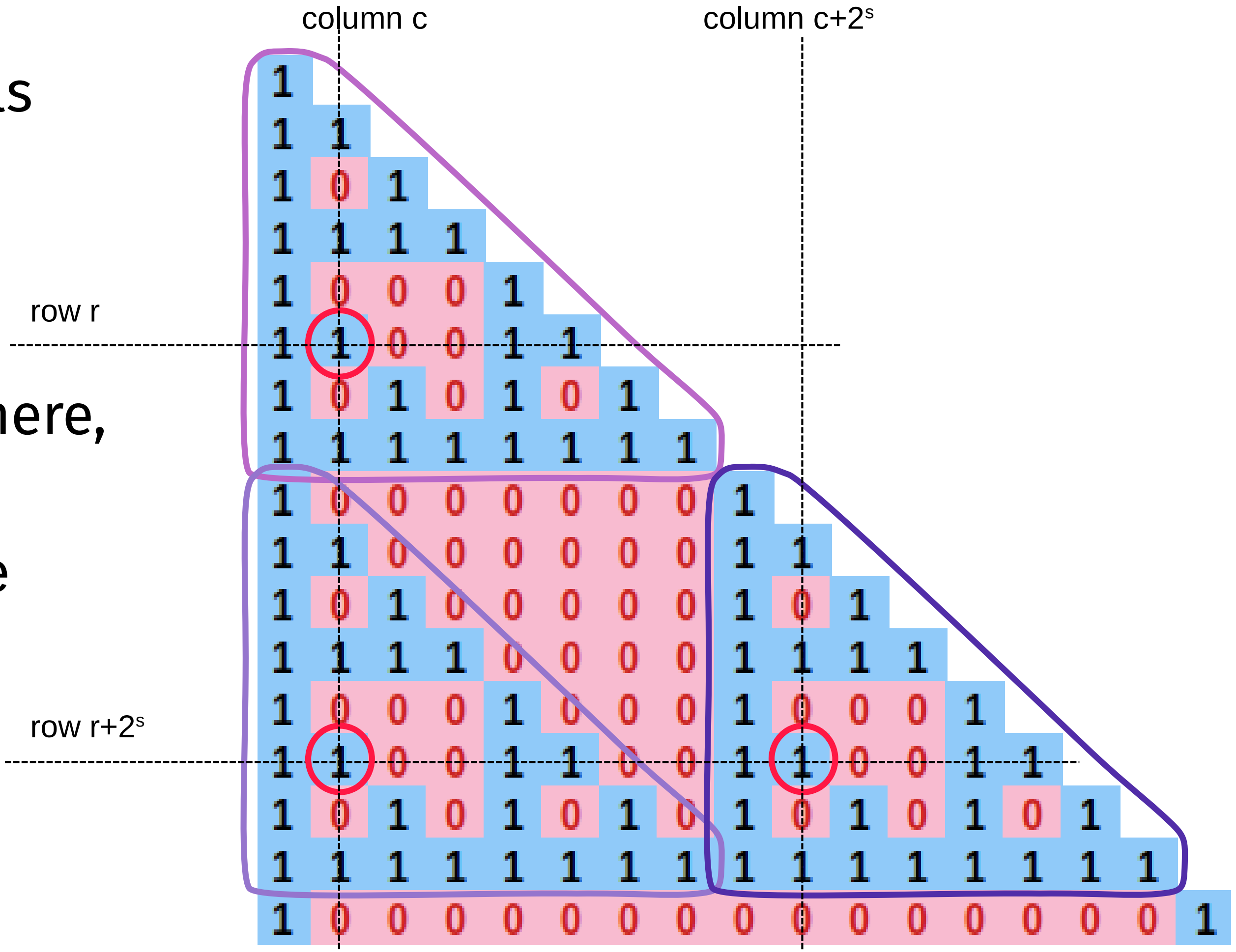
Lucas again. The binomials

$$\binom{r}{c}, \binom{r+2^s}{c} \text{ and } \binom{r+2^s}{c+2^s}$$

have the same parity.

We can observe this fact here, with $r=5$, $c=2$ and $2^s=8$.

The upper “triangle”, made of 8 rows, gets a copy below …
and another one below and to the right.



An insight into the Lucas theorem (3)

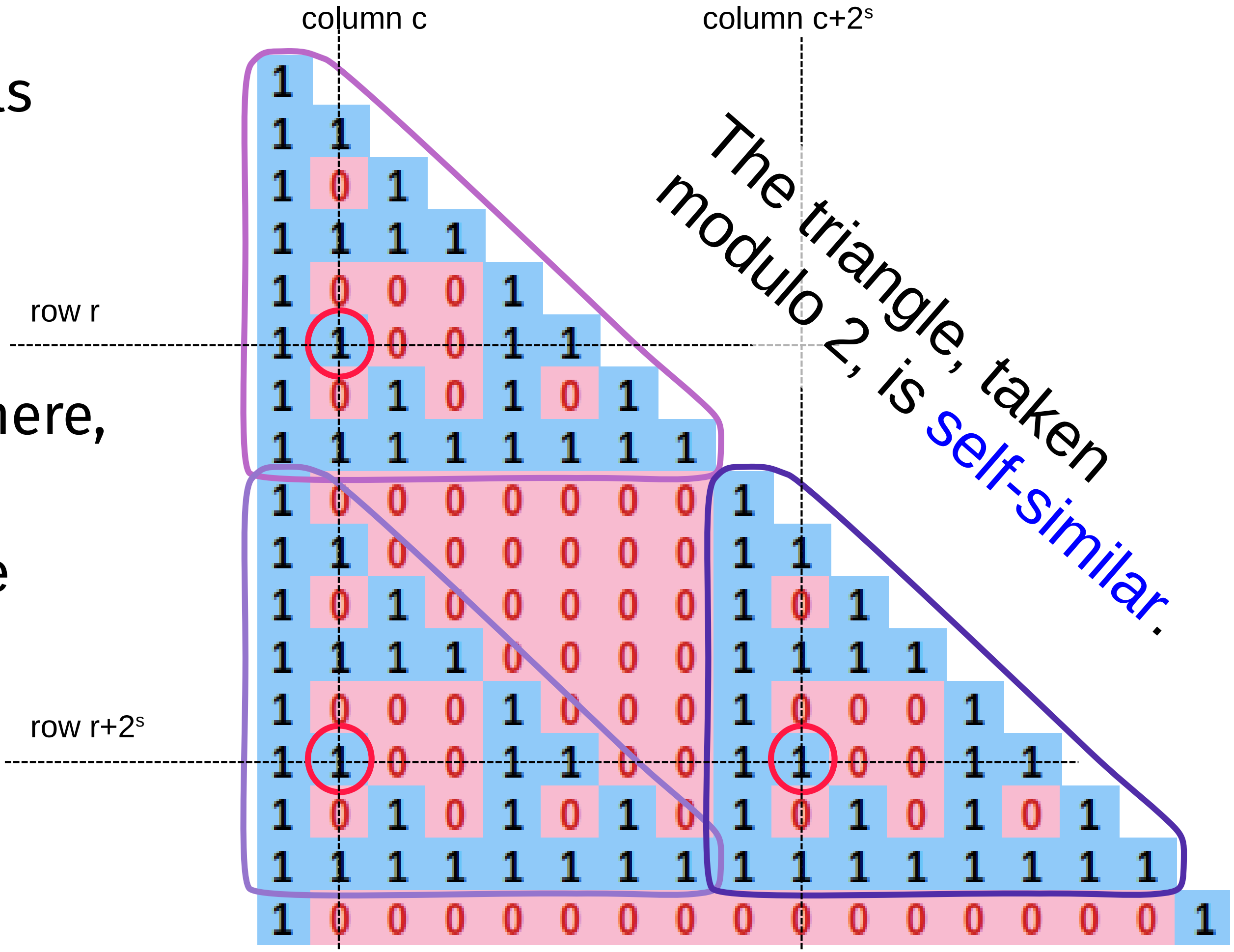
Lucas again. The binomials

$$\binom{r}{c},\ \binom{r+2^s}{c} \text{ and } \binom{r+2^s}{c+2^s}$$

have the same parity.

We can observe this fact here, with $r=5$, $c=2$ and $2^s=8$.

The upper “triangle”, made of 8 rows, gets a copy below …
and another one below and to the right.



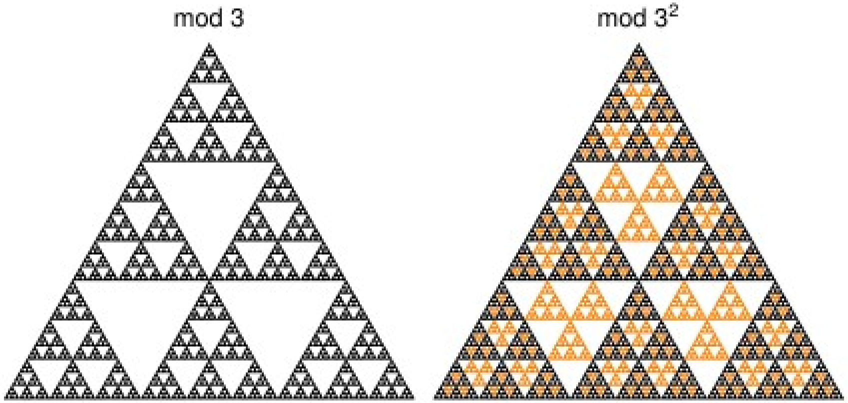
Finally

- Tom Bannink, Harry Buhrman in *“Quantum Pascal’s Triangle and Sierpinski’s carpet”* (2017)

  consider Pascal’s triangle modulo non-prime moduli

  https://arxiv.org/pdf/1708.07429.pdf

  before applying these ideas to quantum computing.



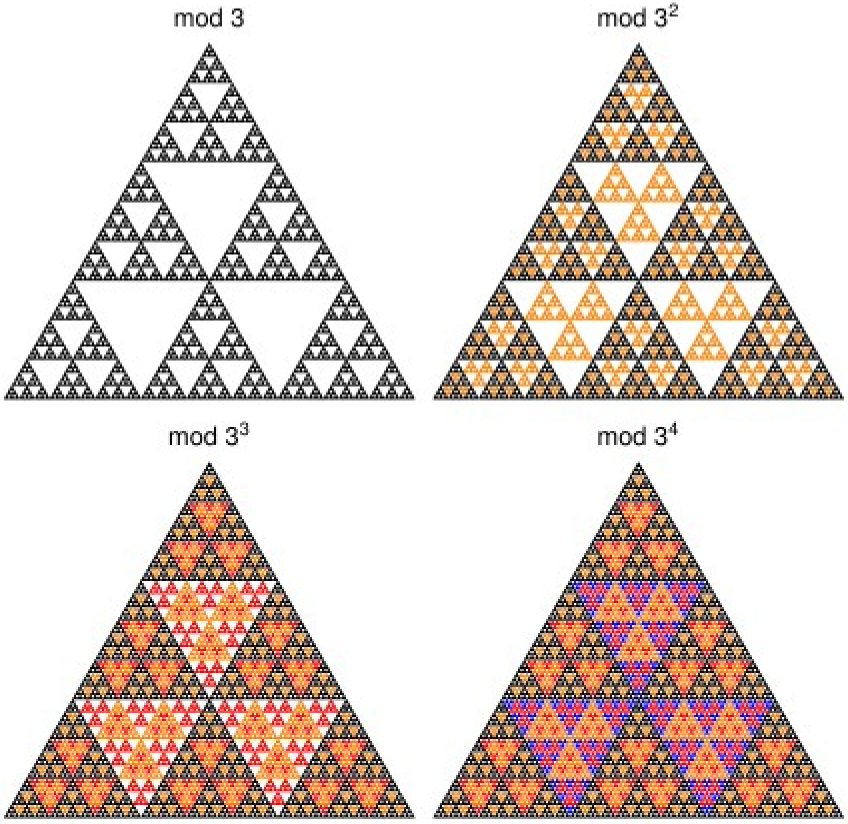
Finally

- Tom Bannink, Harry Buhrman in *“Quantum Pascal’s Triangle and Sierpinski’s carpet”* (2017)

  consider Pascal’s triangle modulo non-prime moduli

  https://arxiv.org/pdf/1708.07429.pdf

  before applying these ideas to quantum computing.
- Here art & math are meeting.

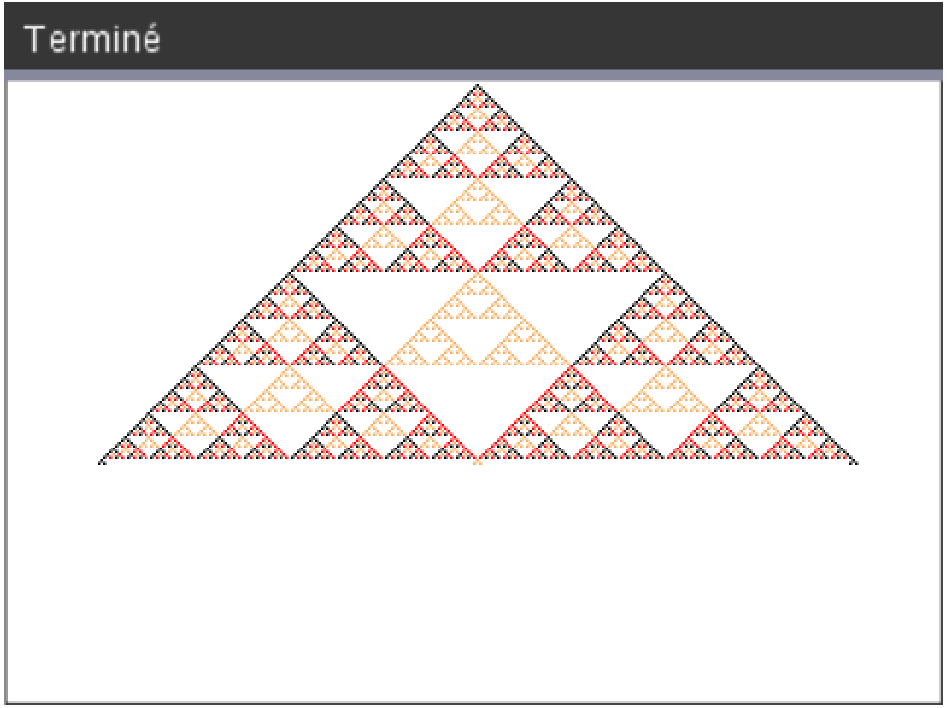


Let's play … modulo 4

*Sierpinski4.py 3/17

```
from ti_draw import *
def pt(i,j,c):
  if c==0: set_color(255,255,255)
  elif c==1: set_color(0,0,0)
  elif c==2: set_color(245,176,99)
  else: set_color(255,0,0)
  plot_xy(160-i+2*j,1+i,7)
def t(p):
  clear()
  n=p+1 ; L=[0]*n ; L[0]=1
  for i in range(n):
    for j in range(i,-1,-1):
      if j>0:
        L[j]=L[j-1]+L[j]
      pt(i,j,L[j]%4)
```

Code : `Sierpinski.tns / SIRPNSKI.8xv`



References

[1] Blaise Pascal, *Traité du triangle arithmétique* (1665). https://gallica.bnf.fr/ark:/12148/btv1b86262012

[2] Wikipedia, *Pascal's Triangle*. https://en.wikipedia.org/wiki/Pascal's_triangle

[3] Wikipedia, *Wacław_Sierpiński*. https://en.wikipedia.org/wiki/Wacław_Sierpiński

[4] Wikipedia, *Édouard Lucas.* https://fr.wikipedia.org/wiki/Édouard_Lucas

[5] Édouard Lucas, *Sur les congruences des nombres eulériens et des coefficients différentiels des fonctions trigonométriques suivant un module premier,* Bull. Soc. Math. France, 6 (1878), pp. 49-54. http://www.numdam.org/articles/10.24033/bsmf.127

[6] N.J. Fine, Binomial coefficients modulo a prime, Amer. Math. Monthly 54 (1947), pp. 589-592

[7] Stephen Wolfram, Geometry of binomial coefficients, Amer. Math. Monthly 91 (1984), pp. 566-571

Bonus : Lucas thorem

In actual notations, Édouard Lucas theorem can be stated as :

Theorem. Let A, B be integers, with $0 \le B \le A$, and p a prime.
Write A and B in p-adic notation as

$$A = a_k p^k + \cdots + a_1 p + a_0, \text{ and } B = b_k p^k + \cdots + b_1 p + b_0$$

where $0 \le a_i, b_i < p$ and $a_k \ne 0$. Then

$$\binom{A}{B} \equiv \binom{a_k}{b_k}\binom{a_{k-1}}{b_{k-1}} \cdots \binom{a_1}{b_1}\binom{a_0}{b_0} \bmod p$$

Corollary. If $2^n > a > b$, then $\binom{2^n + a}{b} \equiv \binom{2^n + a}{2^n + b} \equiv \binom{a}{b} \bmod 2.$