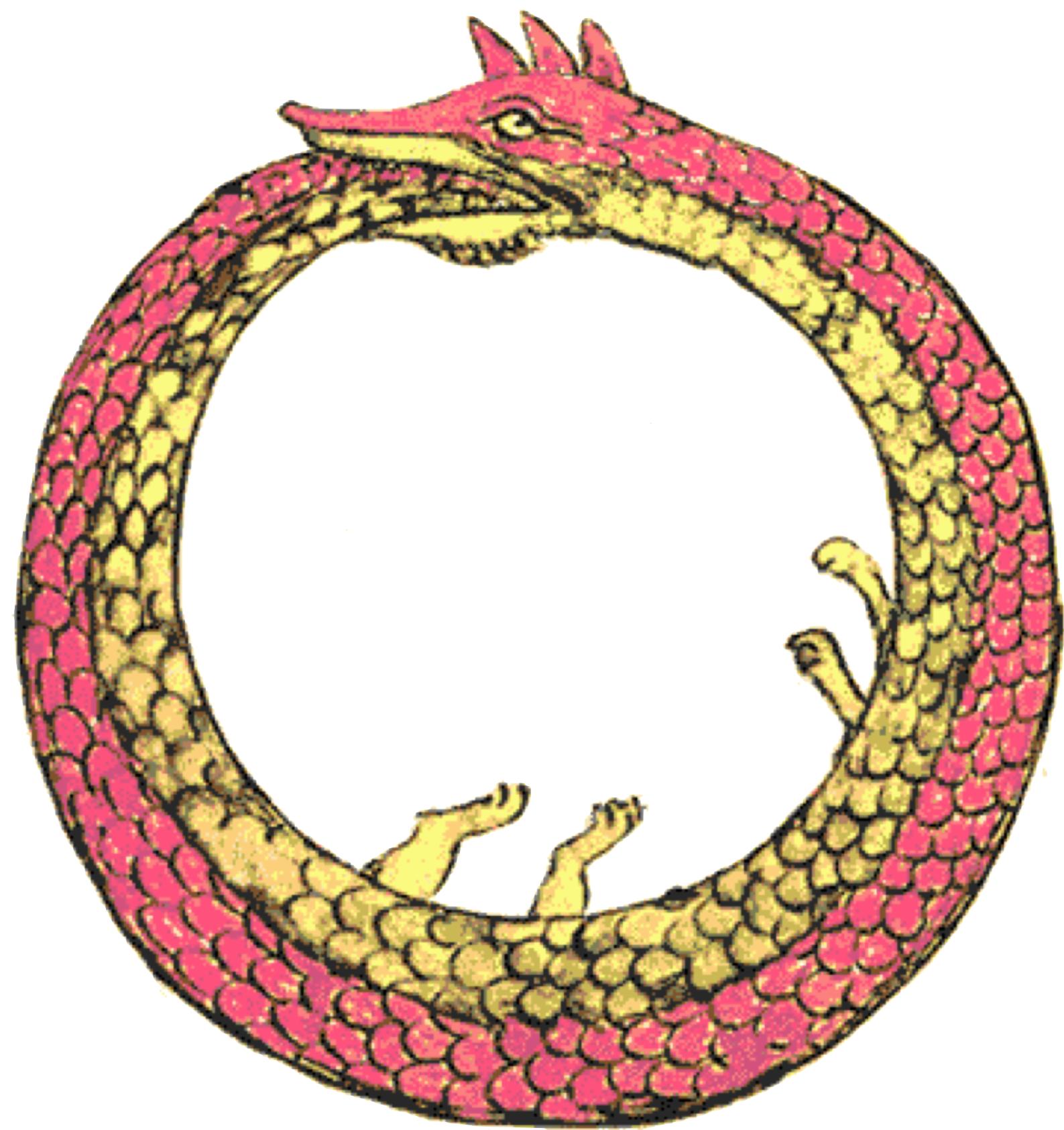
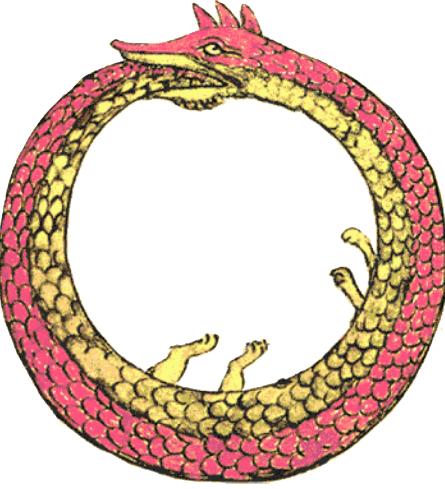


# Récursivité

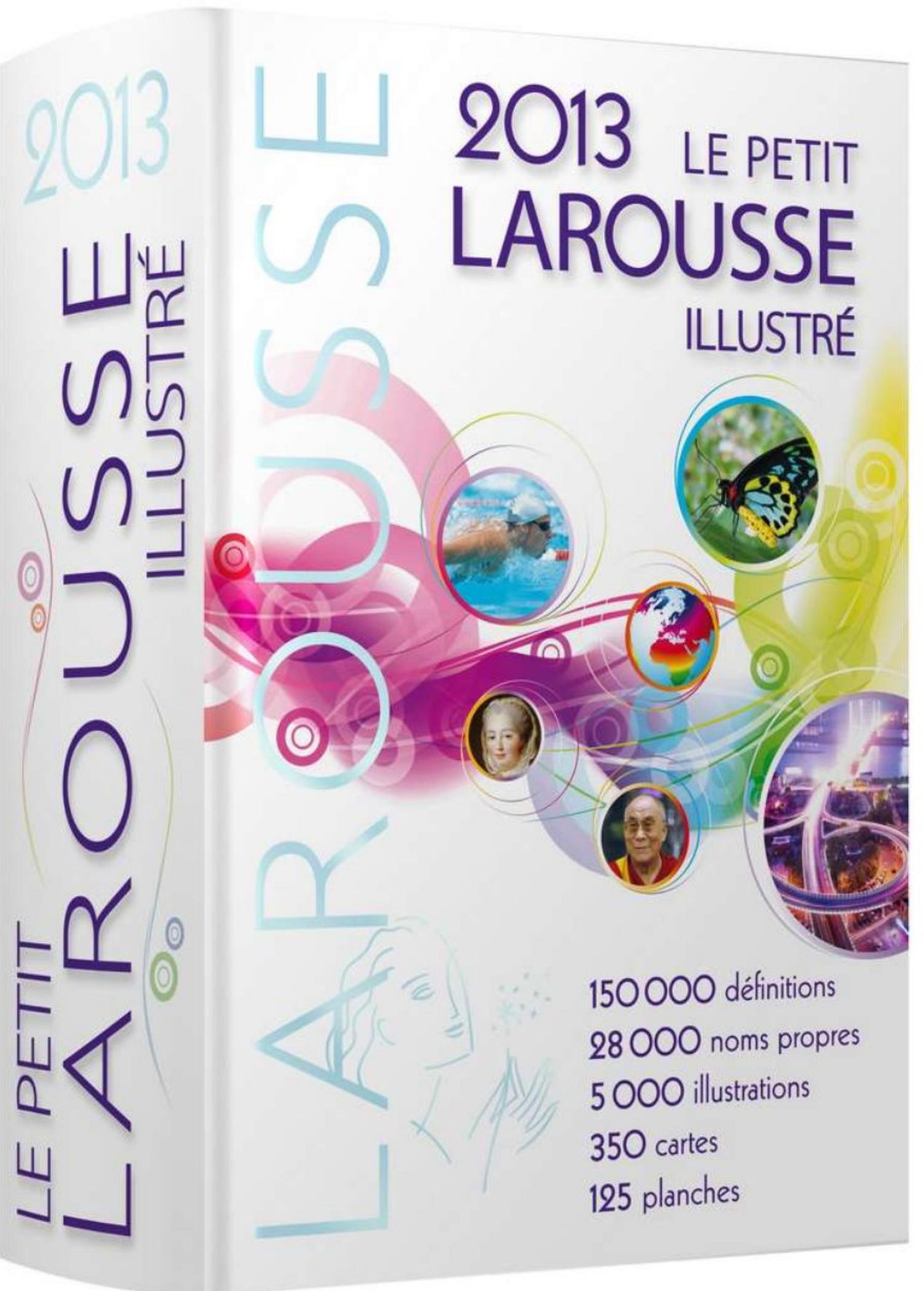


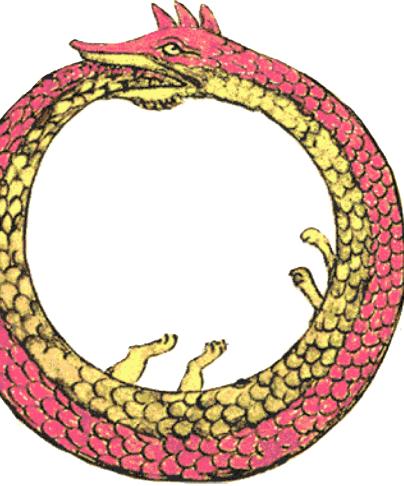
# Définition



**Récursif, récursive**, adjetif, (anglais recursive, du latin recursum, de recurrere, revenir en arrière)

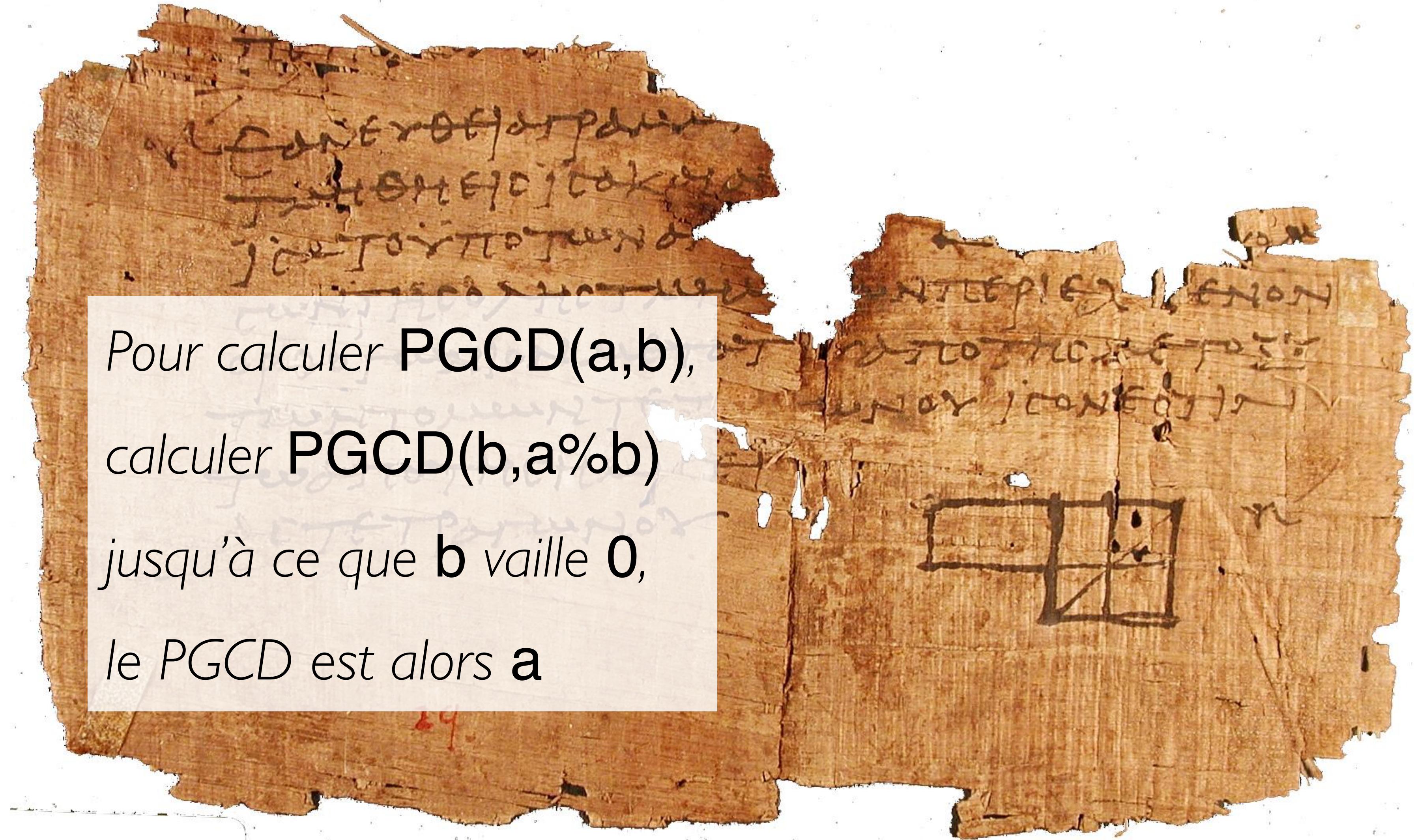
- Se dit d'une règle ou d'un élément doués de récursivité (grammaire, mathématique)
- Se dit d'un programme informatique organisé de manière telle qu'il puisse se rappeler lui-même, c'est-à-dire demander sa propre exécution au cours de son déroulement



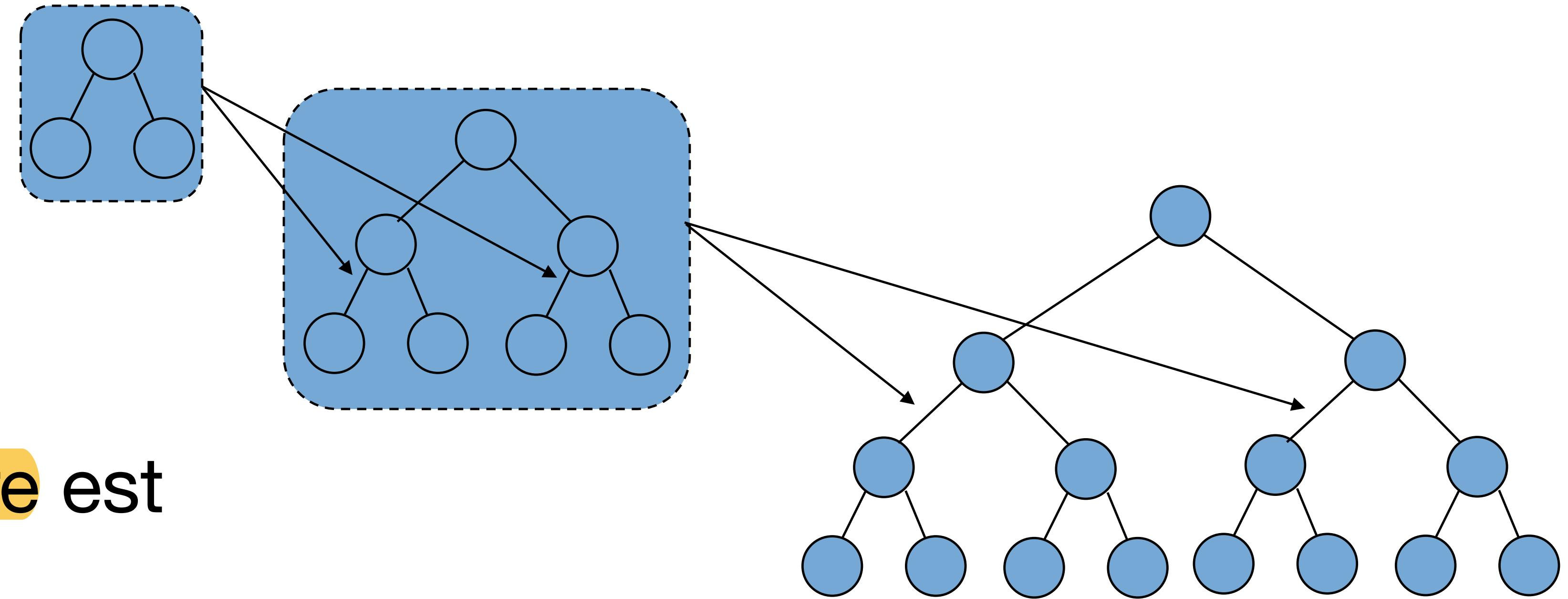


# Exemple : Euclide

Pour calculer  $\text{PGCD}(a,b)$ ,  
calculer  $\text{PGCD}(b,a\%b)$   
jusqu'à ce que  $b$  vaille 0,  
le PGCD est alors  $a$



# Structure récursive

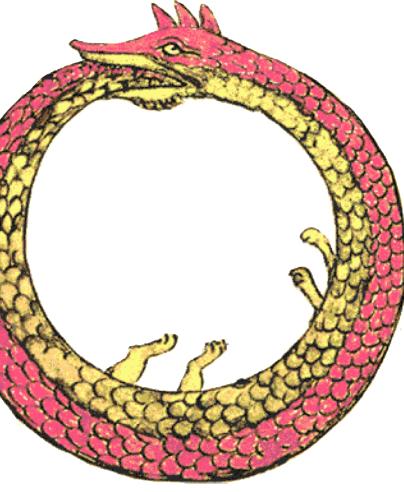


# Un arbre binaire est

- soit un arbre vide,
  - soit une structure composée d'une racine et de deux enfants qui sont eux-même des arbres binaires.

## 2.1. Factorielle

$$n! = \prod_{i=1}^n i$$

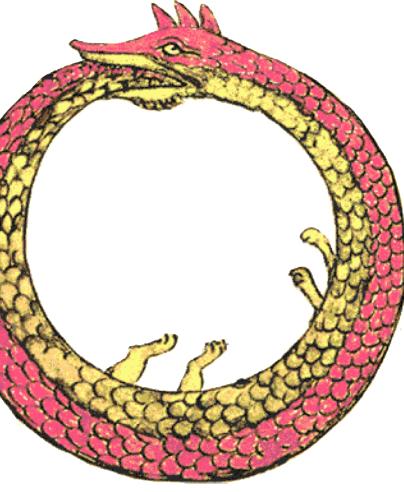


# Factorielle

$$n! = \prod_{i=1}^n i = 1 \times 2 \times 3 \dots \times (n-1) \times n$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \text{ ou } 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

```
fonction factorielle(n)
  si n vaut 0 alors
    retourner 1
  sinon
    retourner n x factorielle(n-1)
  fin si
```



# Traçons l'exécution ...

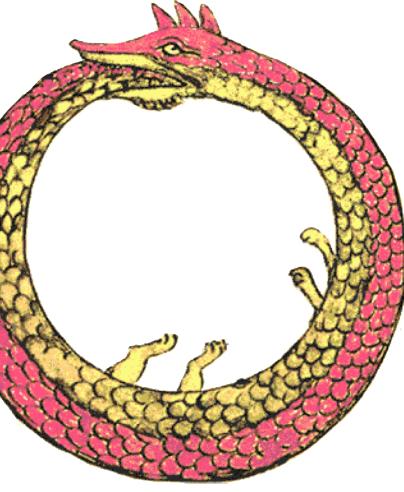
```
fonction factorielle(n)
  si n vaut 0 alors
    retourner 1
  sinon
    retourner n x factorielle(n-1)
  fin si
```

**Calculer** 4!

4 != 0

*Calculer* 3!

**retourner** 4 x 3!



# Traçons l'exécution ...

```
fonction factorielle(n)
    si n vaut 0 alors
        retourner 1
    sinon
        retourner n x factorielle(n-1)
    fin si
```

**Calculer 4!**

4 != 0

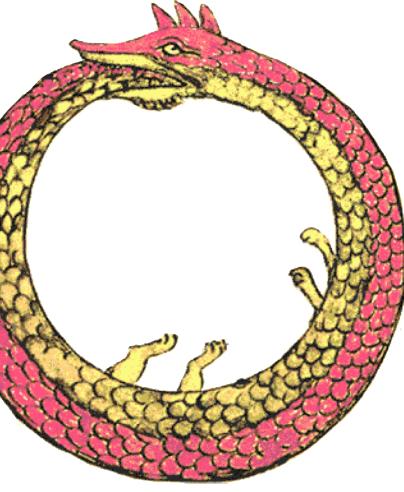
*Calculer 3!*

3 != 0

*Calculer 2!*

**retourner 3 x 2!**

**retourner 4 x 3!**



# Traçons l'exécution ...

```
fonction factorielle(n)
  si n vaut 0 alors
    retourner 1
  sinon
    retourner n x factorielle(n-1)
  fin si
```

**Calculer** 4!

4 != 0

*Calculer* 3!

3 != 0

*Calculer* 2!

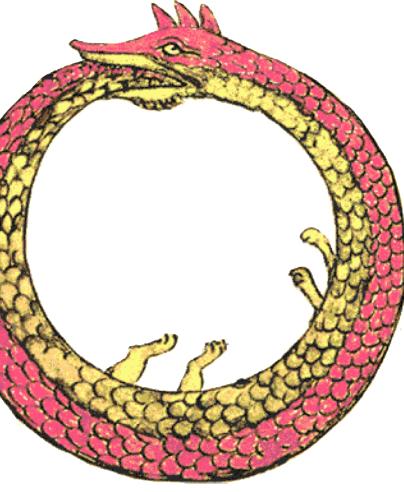
2 != 0

*Calculer* 1!

**retourner** 2 x 1!

**retourner** 3 x 2!

**retourner** 4 x 3!



# Traçons l'exécution ...

```
fonction factorielle(n)
    si n vaut 0 alors
        retourner 1
    sinon
        retourner n x factorielle(n-1)
    fin si
```

**Calculer 4!**

$4 != 0$

*Calculer 3!*

$3 != 0$

*Calculer 2!*

$2 != 0$

*Calculer 1!*

$1 != 0$

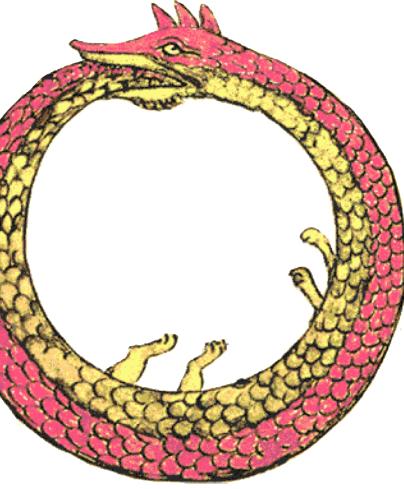
*Calculer 0!*

**retourner 1 x 0!**

**retourner 2 x 1!**

**retourner 3 x 2!**

**retourner 4 x 3!**



# Traçons l'exécution ...

```
fonction factorielle(n)
    si n vaut 0 alors
        retourner 1
    sinon
        retourner n x factorielle(n-1)
    fin si
```

**Calculer 4!**

4 != 0

*Calculer 3!*

3 != 0

*Calculer 2!*

2 != 0

*Calculer 1!*

1 != 0

*Calculer 0!*

0 == 0

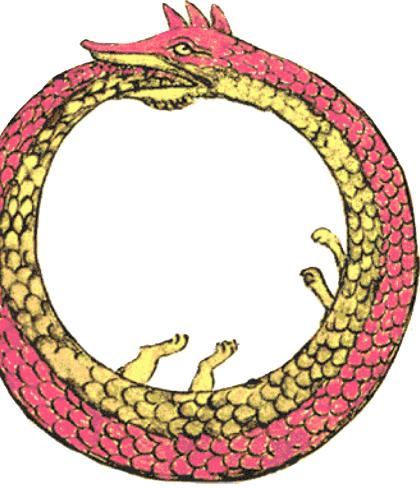
**retourner 1**

**retourner 1 x 0!**

**retourner 2 x 1!**

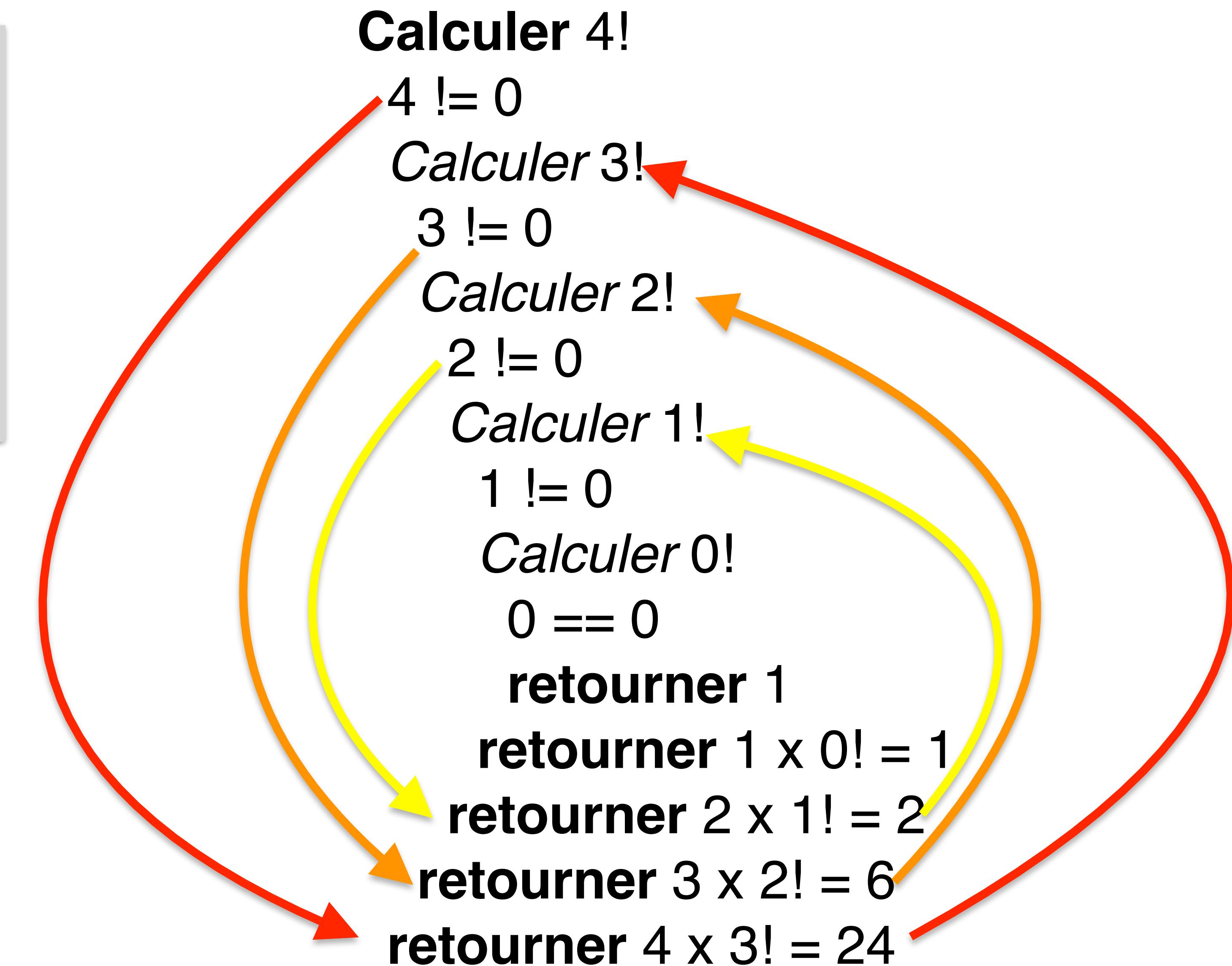
**retourner 3 x 2!**

**retourner 4 x 3!**

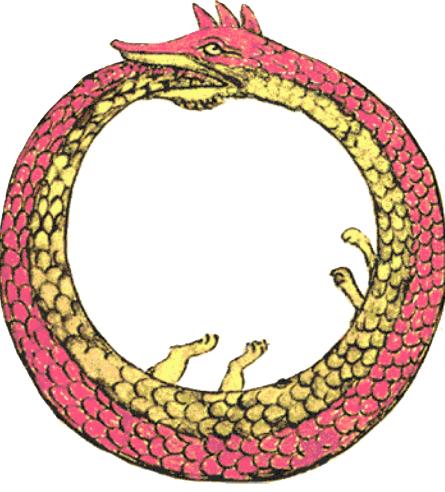


# Traçons l'exécution ...

```
fonction factorielle(n)
    si n vaut 0 alors
        retourner 1
    sinon
        retourner n x factorielle(n-1)
    fin si
```



# Cas trivial, cas général



**fonction factorielle(n)**

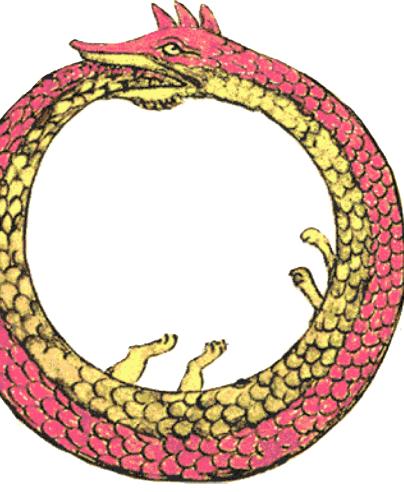
Paramètre de  
récursivité

**si n vaut 0 alors**  
**retourner 1**

**Cas trivial**

**sinon**  
**retourner**  $n \times \text{factorielle}(n-1)$   
**fin si**

**Cas général**



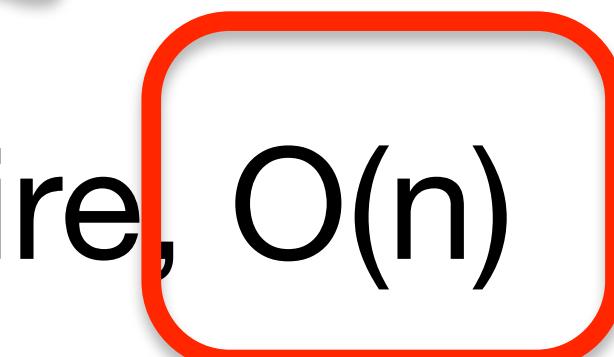
# Complexité ?

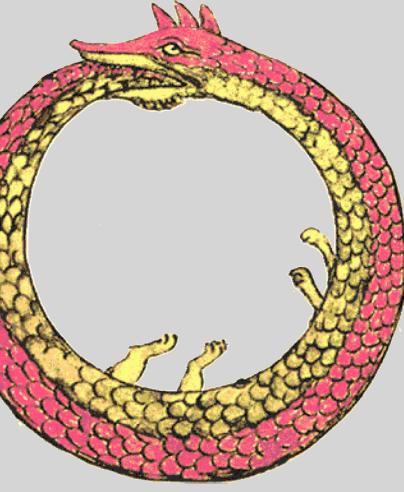
- Combien d'appels récursifs sont effectués ?

```
fonction factorielle(n)
    si n vaut 0 alors
        retourner 1
    sinon
        retourner n x factorielle(n-1)
    fin si
```



- Complexité linéaire,  $O(n)$





# Exercice S-2.1

- Quelles sont les complexités de ces fonctions récursives ?

```
int f1(unsigned n)
{
    if(n == 0)
        return -1;
    else
        return 1 + f1(n/2);
}
```

$\frac{n}{2}$

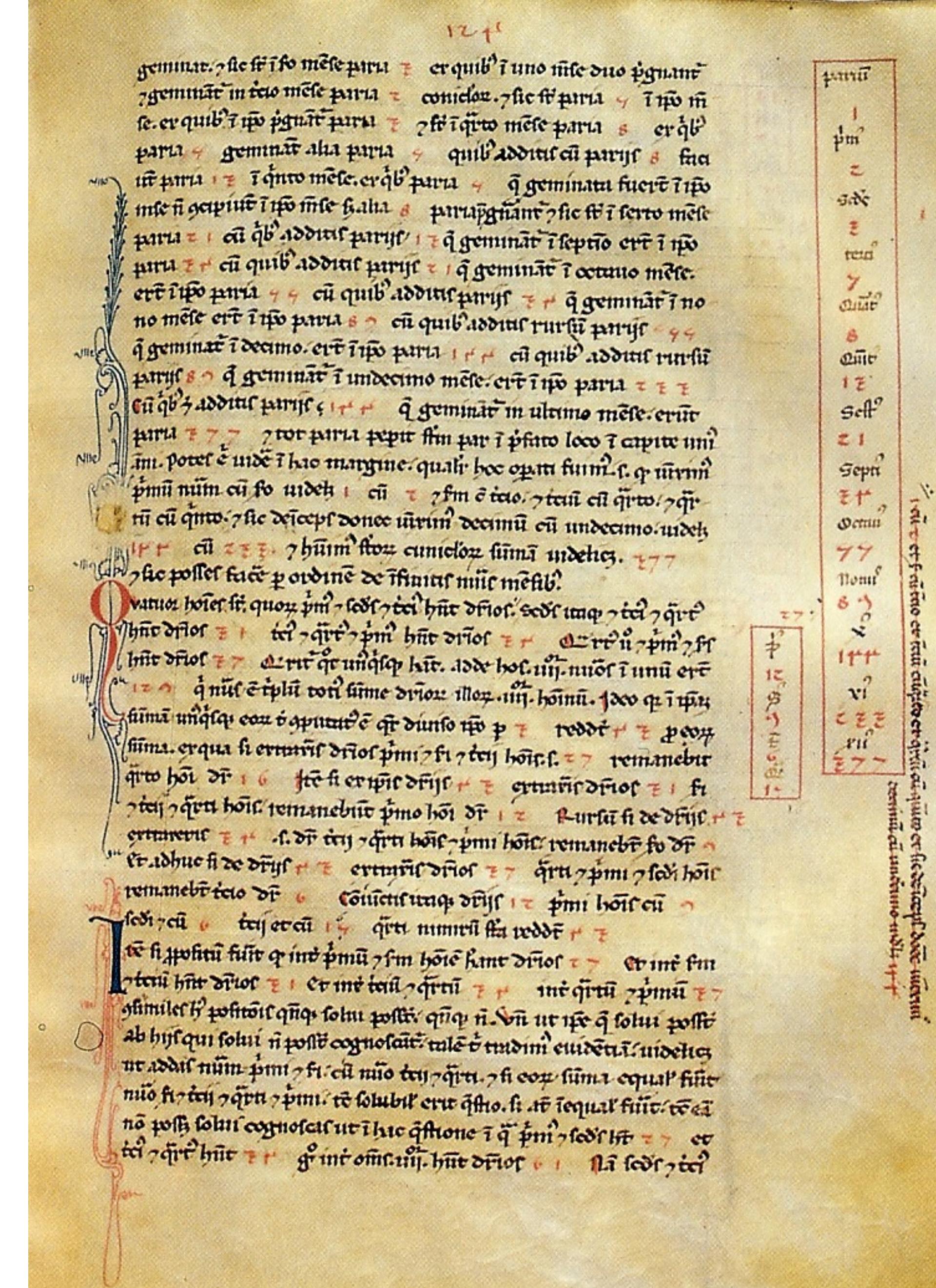
```
int f2(double n) {
    if(n <= 1)
        return 1;
    else
        return 1 + f2(n-2*sqrt(n)+1);
}
```

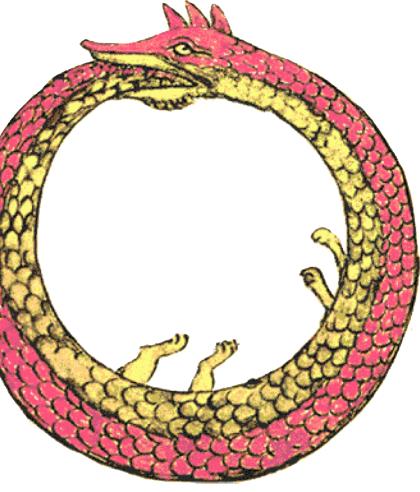
## 2.2. Fibonacci



# Leonardo Fibonacci

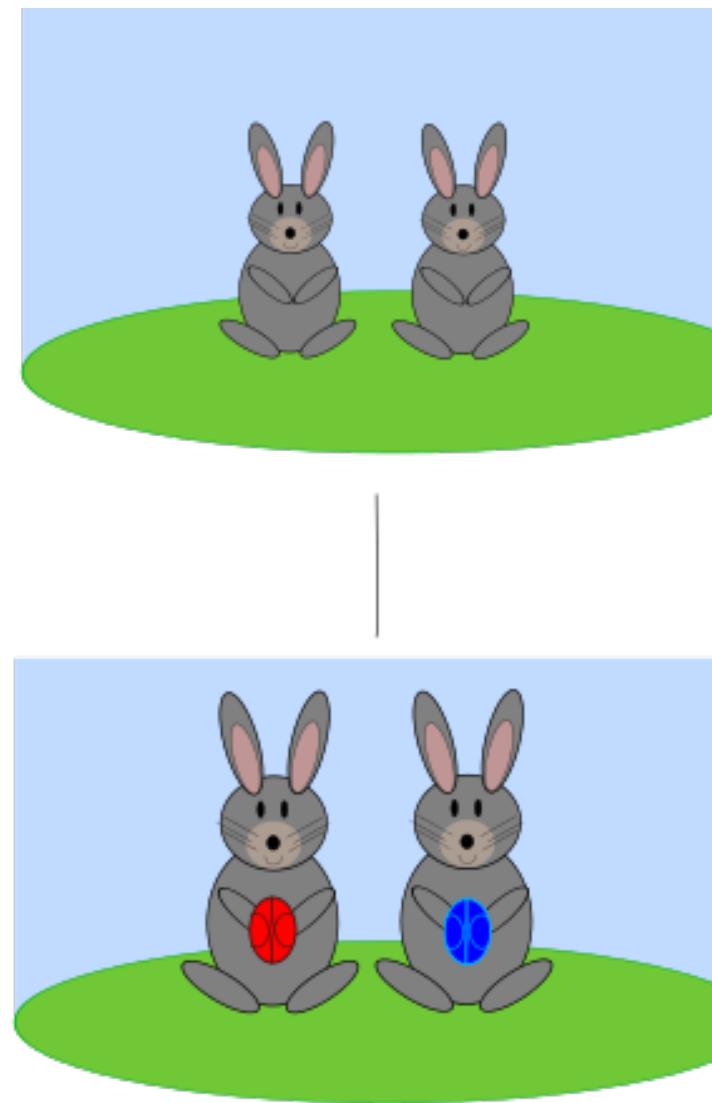
- Il s'intéresse en 1202 à la croissance d'une population de lapins idéalisée.
- On commence avec un couple de lapins nouveaux nés.
- A partir de l'âge de 2 mois, un couple de lapin engendre un couple de lapins tous les mois.
- Combien de couples de lapins obtient-on après n mois? (les lapins sont immortels)



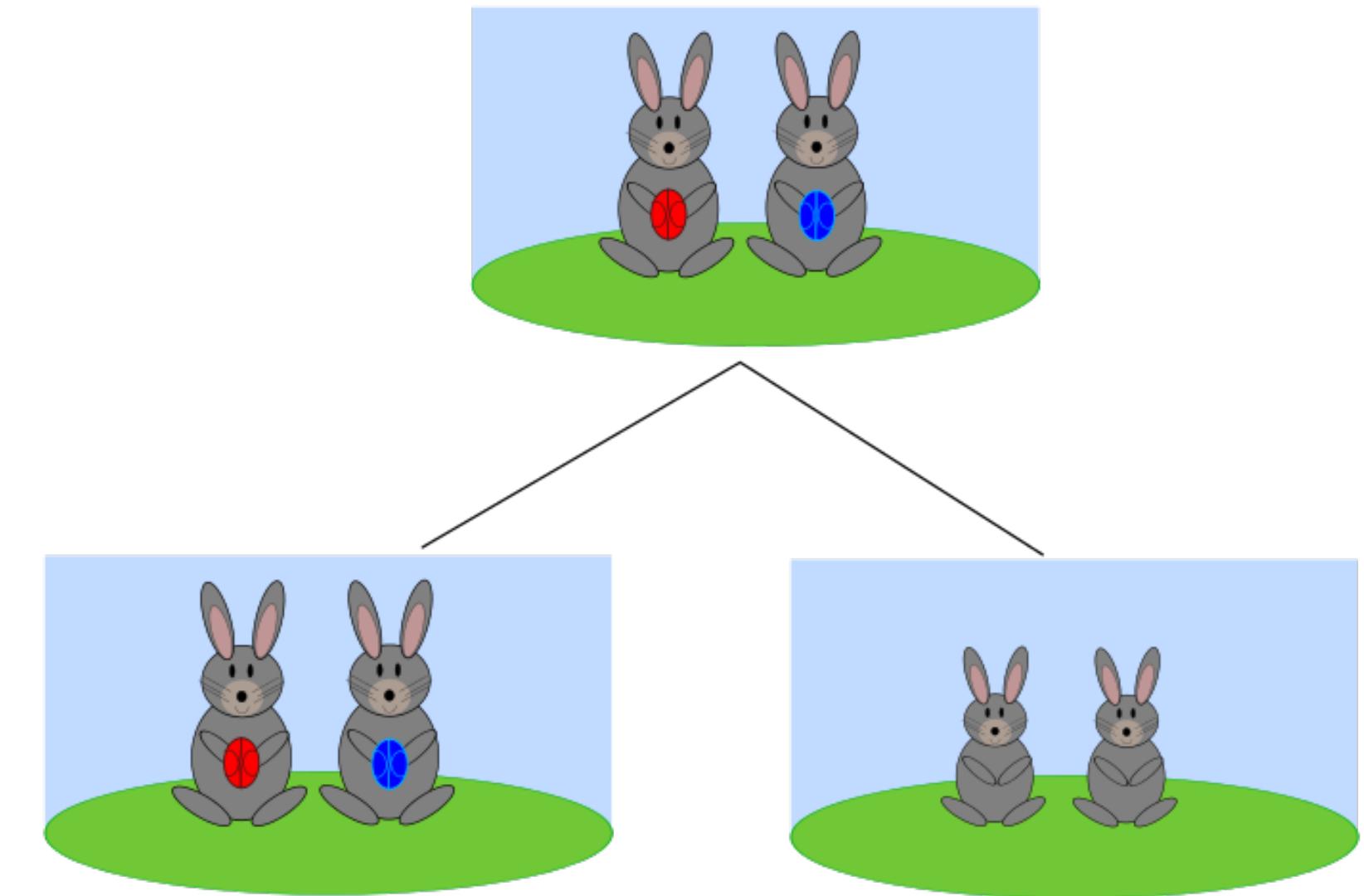


# Graphiquement

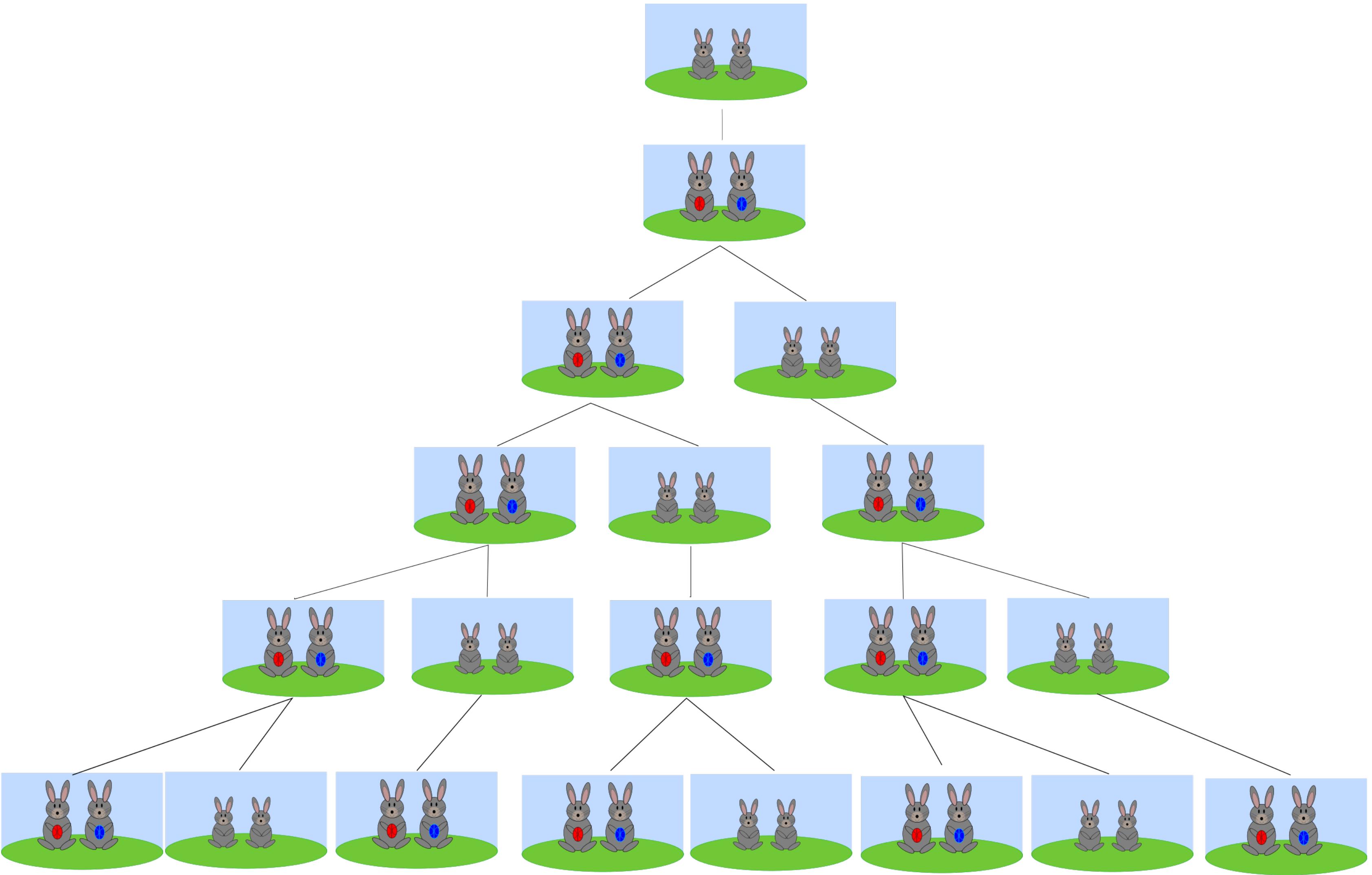
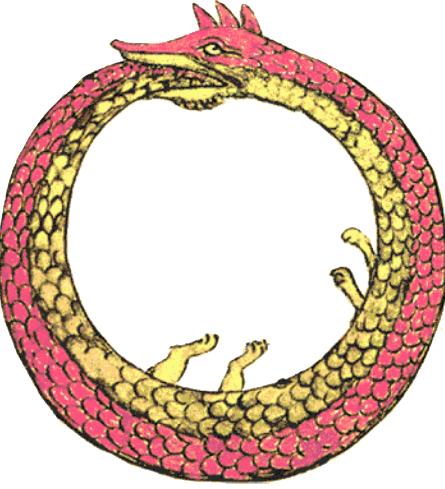
Les enfants deviennent adultes le mois suivant



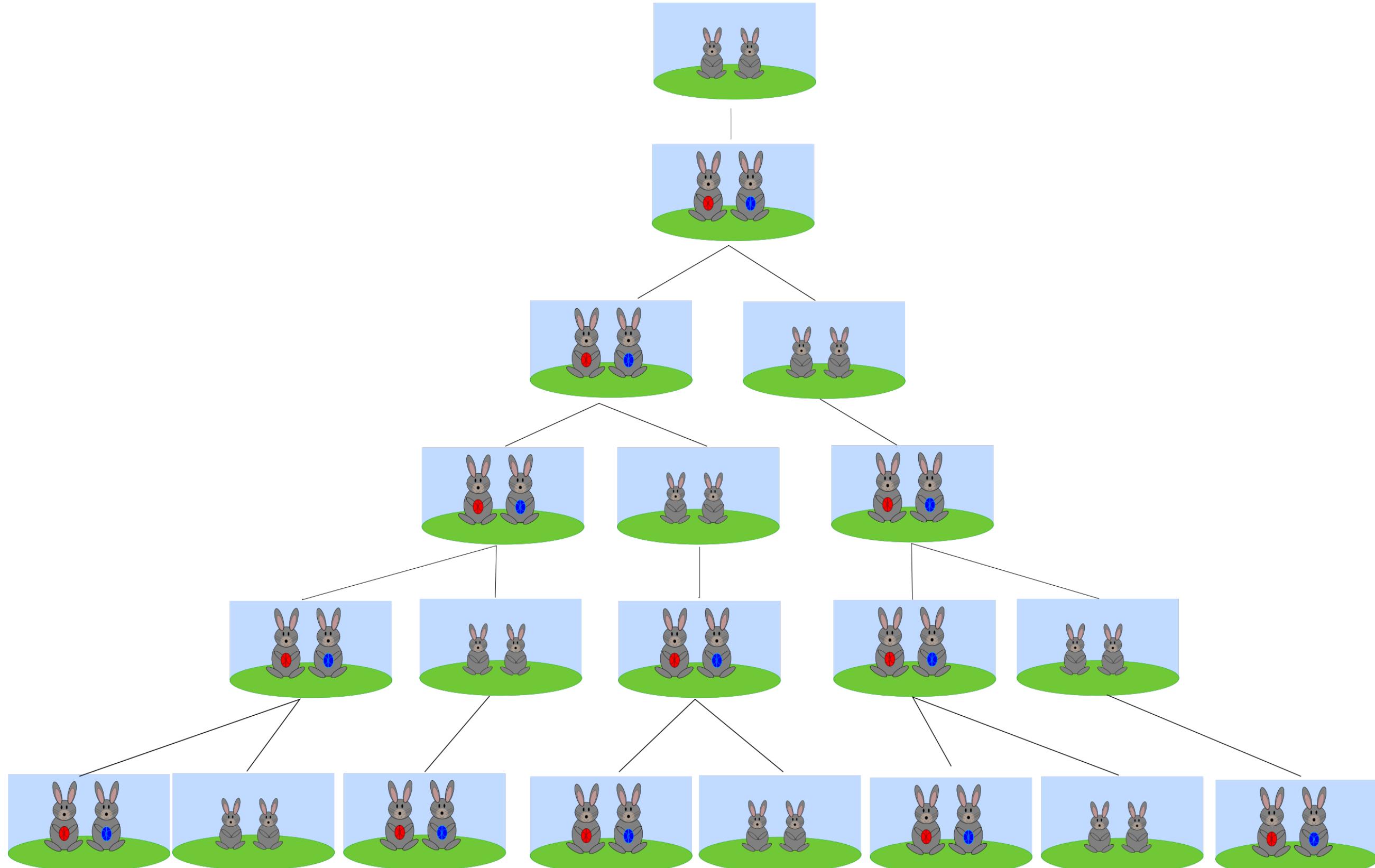
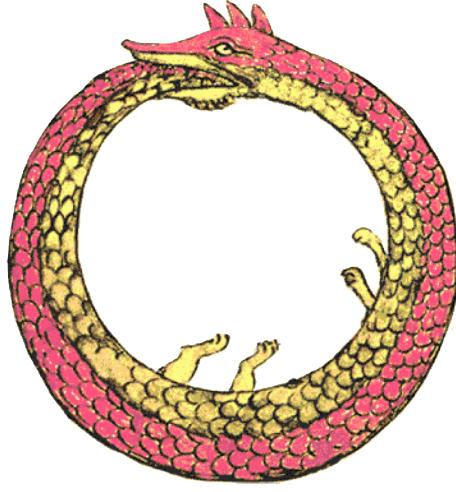
Les adultes font des enfants et survivent tous les mois



# Au fil des générations ...



# Au fil des générations ...



$$\begin{aligned}F(0) &= 0 \\F(1) &= 1 \\F(n) &= F(n-1) + F(n-2)\end{aligned}$$

**fonction  $F(n)$  (version récursive)**

**si  $n$  vaut 0 ou 1 alors**

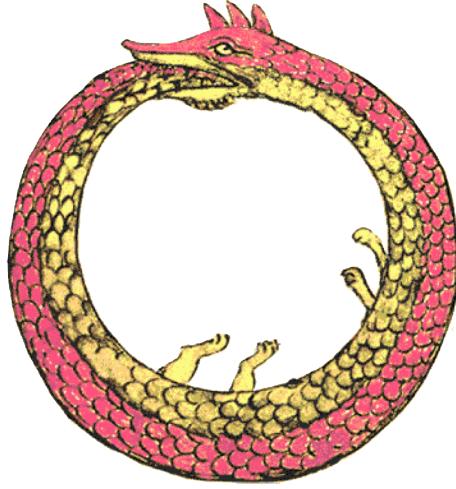
**retourner  $n$**

**sinon**

**retourner  $F(n-1) + F(n-2)$**

**fin si**

# Traçons l'exécution ...



**fonction F(n) (version récursive)**

**si n vaut 0 ou 1 alors**

**retourner n**

**sinon**

**retourner F(n-1) + F(n-2)**

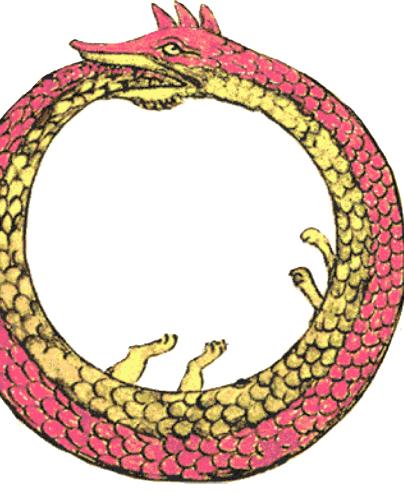
**fin si**

**Calculer F(4)**

*Calculer F(3)*

*Calculer F(2)*

**retourner F(3)+F(2)**



# Traçons l'exécution ...

**fonction F(n) (version récursive)**

**si n vaut 0 ou 1 alors**

**retourner n**

**sinon**

**retourner F(n-1) + F(n-2)**

**fin si**

**Calculer F(4)**

*Calculer F(3)*

*Calculer F(2)*

*Calculer F(1)*

**retourner F(2)+F(1)**

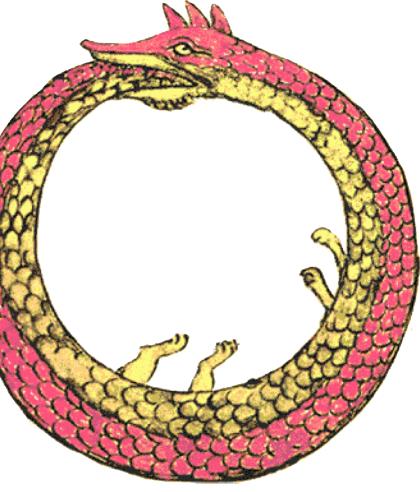
*Calculer F(2)*

*Calculer F(1)*

*Calculer F(0)*

**retourner F(1)+F(0)**

**retourner F(3)+F(2)**



# Traçons l'exécution ...

**fonction F(n) (version récursive)**

**si** n vaut 0 ou 1 **alors**

**retourner** n

**sinon**

**retourner** F(n-1) + F(n-2)

**fin si**

**Calculer** F(4)

*Calculer* F(3)

*Calculer* F(2)

*Calculer* F(1)

*Calculer* F(0)

**retourner** F(1)+F(0)

*Calculer* F(1)

**retourner** 1

**retourner** F(2)+F(1)

*Calculer* F(2)

*Calculer* F(1)

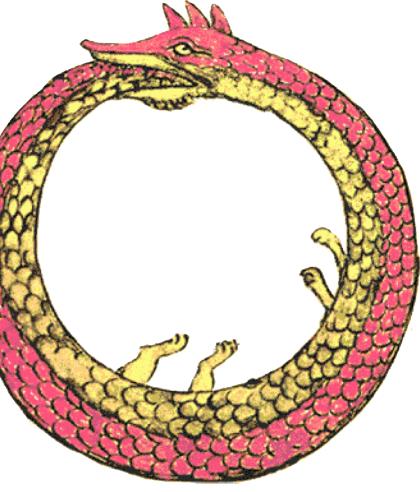
**retourner** 1

*Calculer* F(0)

**retourner** 0

**retourner** F(1) + F(0)

**retourner** F(3)+F(2)



# Traçons l'exécution ...

**fonction F(n) (version récursive)**

**si** n vaut 0 ou 1 **alors**

**retourner** n

**sinon**

**retourner** F(n-1) + F(n-2)

**fin si**

**Calculer** F(4)

*Calculer* F(3)

*Calculer* F(2)

*Calculer* F(1)

**retourner** 1

*Calculer* F(0)

**retourner** 0

**retourner** F(1)+F(0) = 1

*Calculer* F(1)

**retourner** 1

**retourner** F(2)+F(1) = 2

*Calculer* F(2)

*Calculer* F(1)

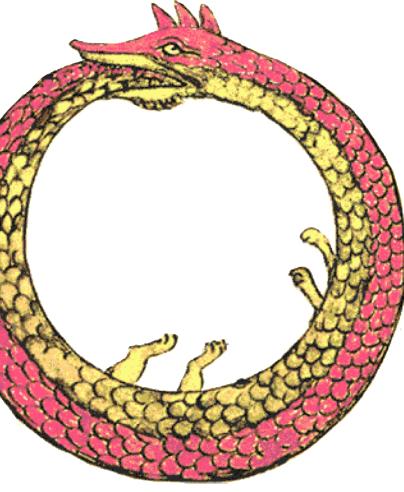
**retourner** 1

*Calculer* F(0)

**retourner** 0

**retourner** F(1) + F(0) = 1

**retourner** F(3)+F(2) = 3



# Arbre des appels

**fonction  $F(n)$  (version récursive)**

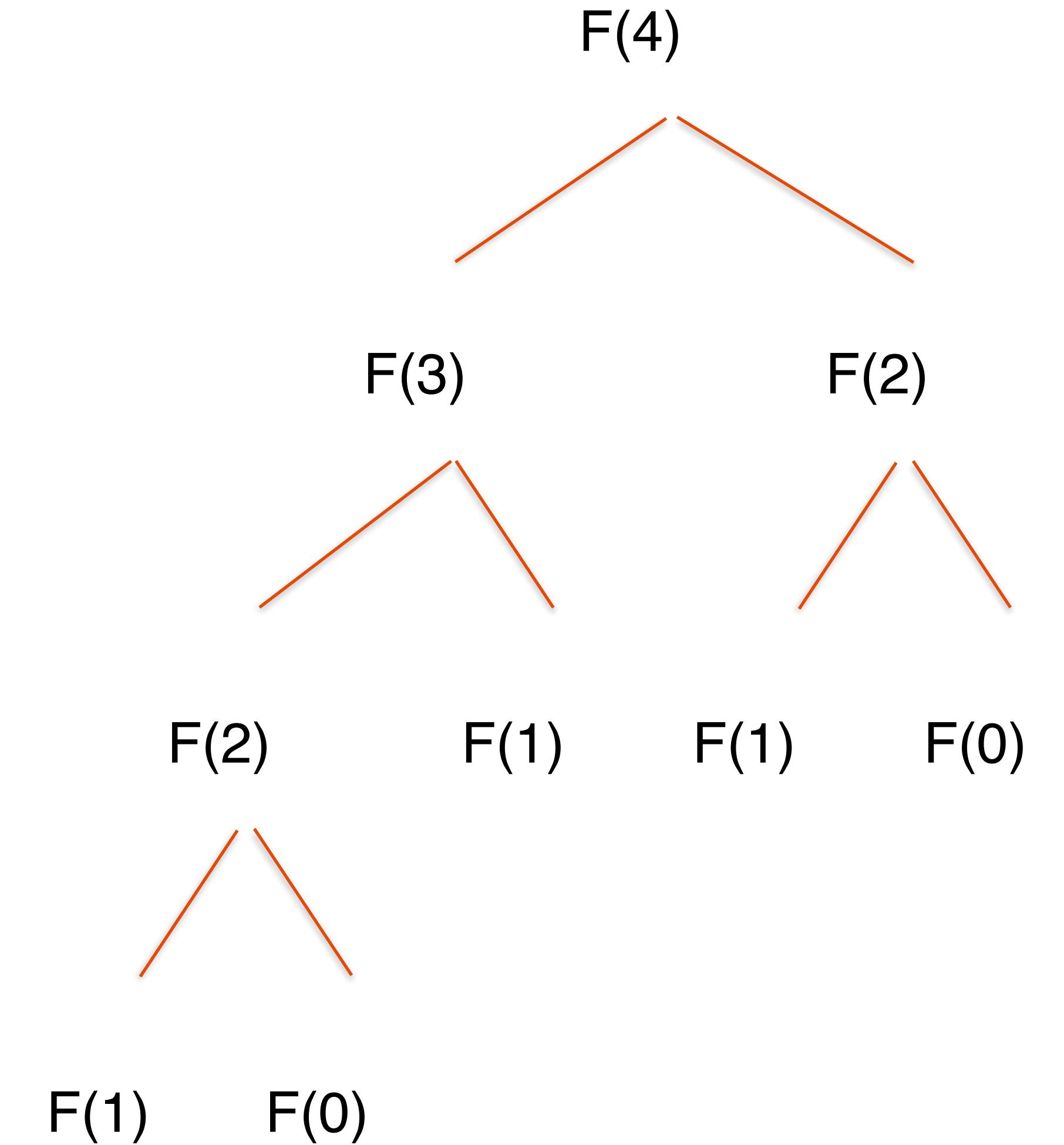
**si  $n$  vaut 0 ou 1 alors**

**retourner  $n$**

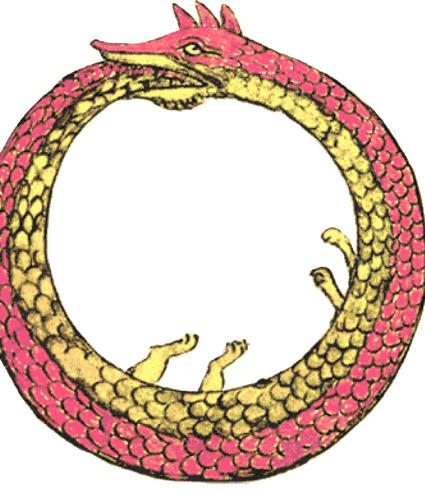
**sinon**

**retourner  $F(n-1) + F(n-2)$**

**fin si**



# Complexité



- Quel est le nombre  $A(n)$  d'appels récursifs effectués pour calculer  $F(n)$ ?

$$A(0) = 0$$

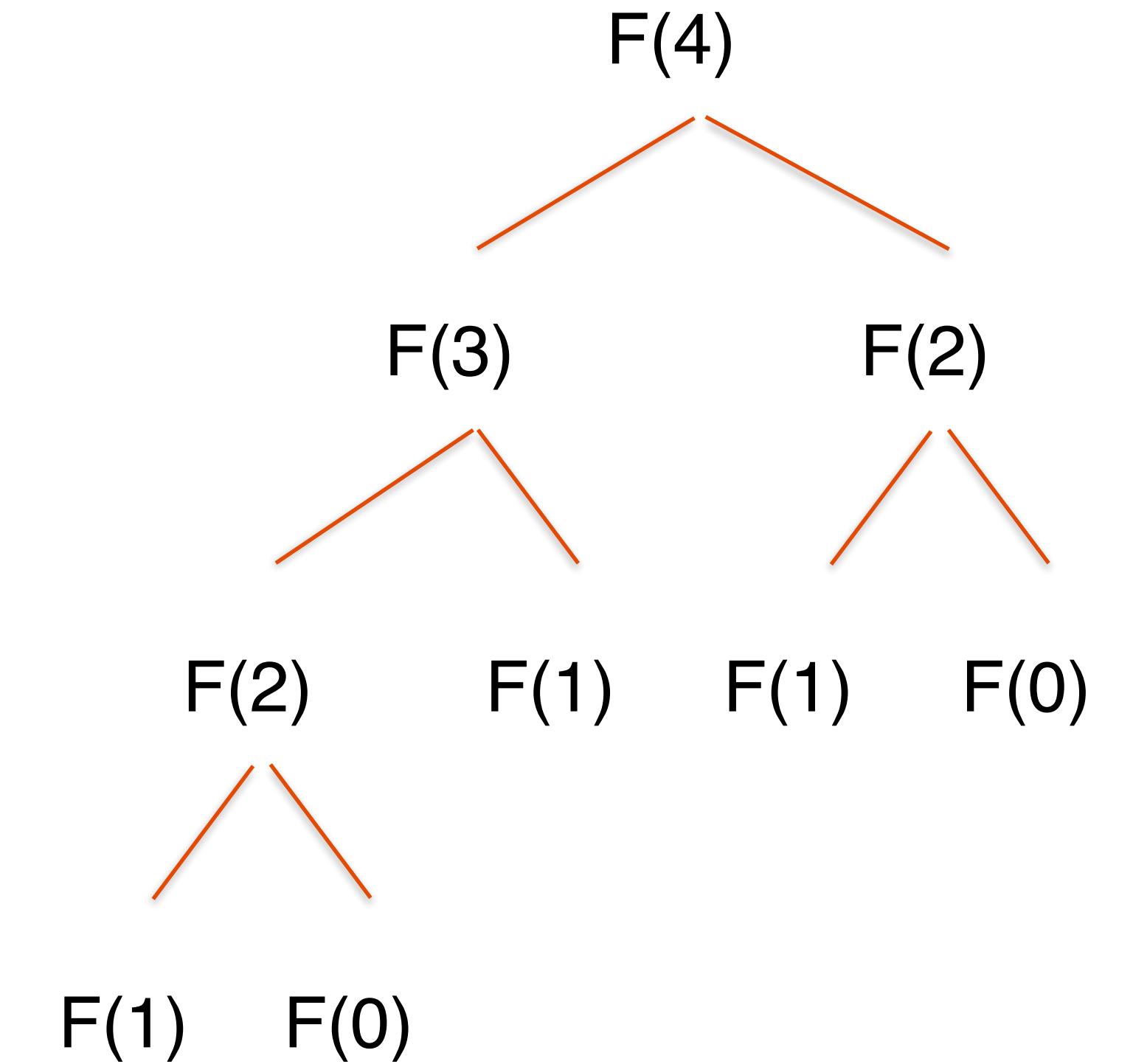
$$A(1) = 0$$

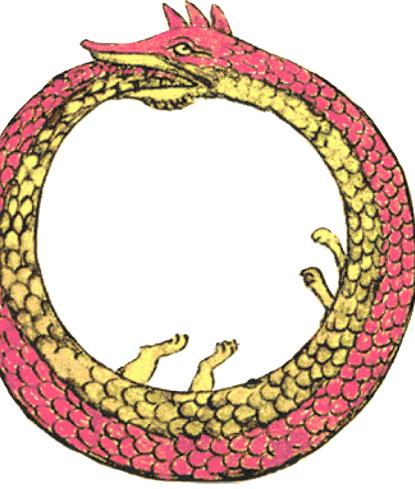
$$A(2) = 2$$

$$A(3) = 2 + A(2) = 4$$

$$A(4) = 2 + A(3) + A(2) = 8$$

$$A(n) = 2 + A(n-1) + A(n-2)$$





# Complexité (2)

n	0	1	2	3	4	5	6	7	8	9
F(n)	0	1	1	2	3	5	8	13	21	33
2*F(n)	0	2	2	4	6	10	16	26	42	66
A(n)	0	0	2	4	8	14	24	40	66	108

On observe la relation suivante :  $A(n) = 2F(n+1) - 2$ . On va la démontrer:

$$\text{Pour } n = 0 : A(0) = 2F(1) - 2 = 2*1 - 2 = 0$$

On la suppose vraie pour  $n$ , on la démontre pour  $n+1$ :

$$\text{On veut montrer que : } A(n + 1) = 2F(n + 2) - 2$$

$$A(n+1) = 2 + A(n) + A(n-1) = 2 + 2F(n+1) - 2 + 2F(n) - 2$$

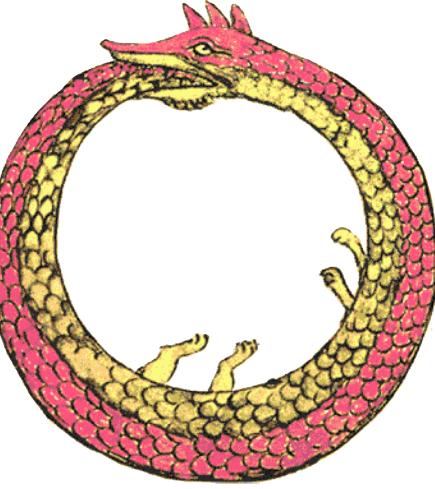
$$= 2((F(n+1) + F(n)) - 2$$

$$= 2F(n+2) - 2$$

$$F(n) = F(n-1) + F(n-2)$$

$$A(n) = 2 + A(n-1) + A(n-2)$$

# Comment croît $F(n)$ ?



- Supposons que  $F(n)$  croisse comme une exponentielle  $\approx x^n$ , que vaut  $x$  ?

- On sait que  $F(n) = F(n-1) + F(n-2)$

$$x^n = x^{n-1} + x^{n-2}$$

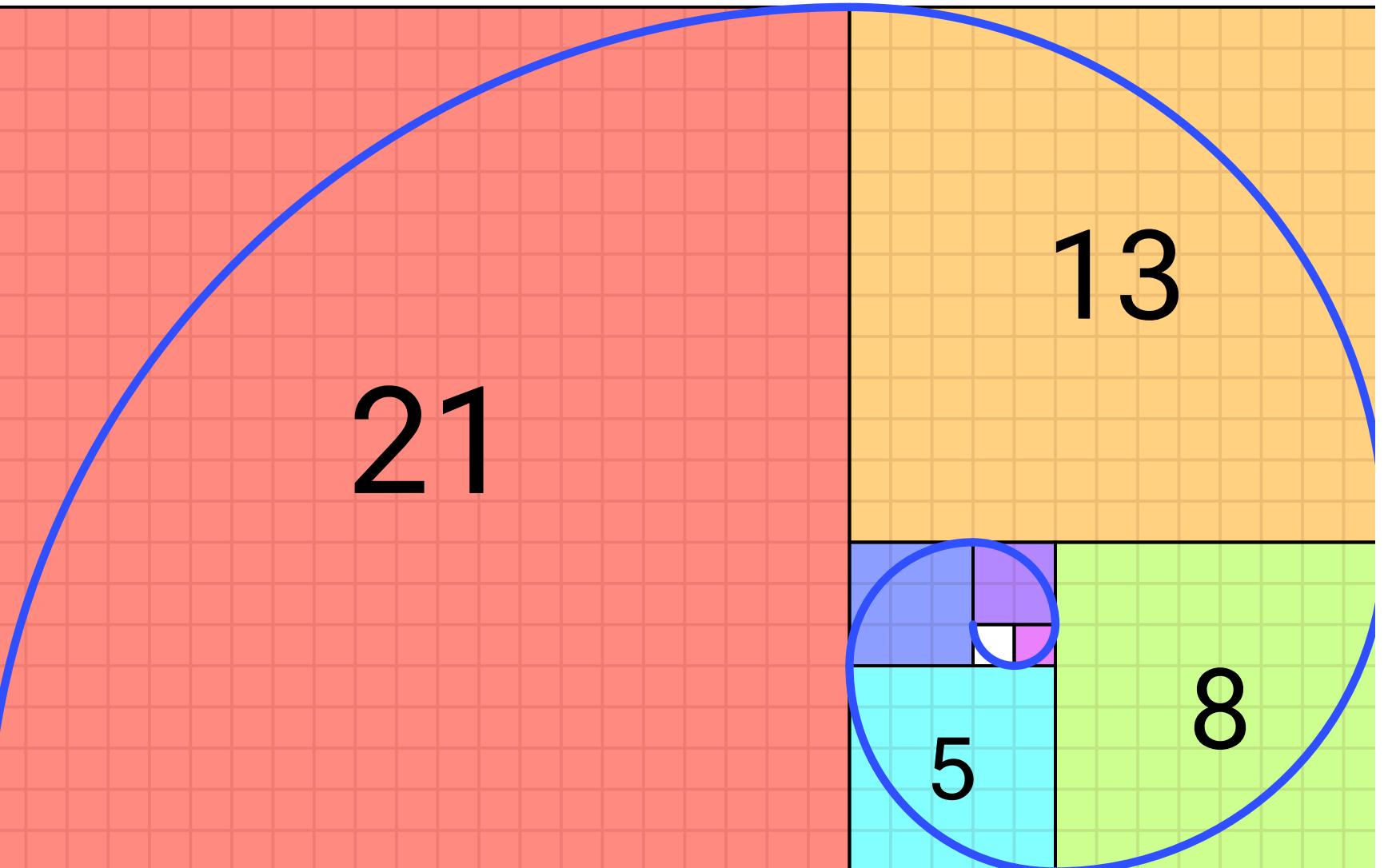
$$x^2 = x + 1$$

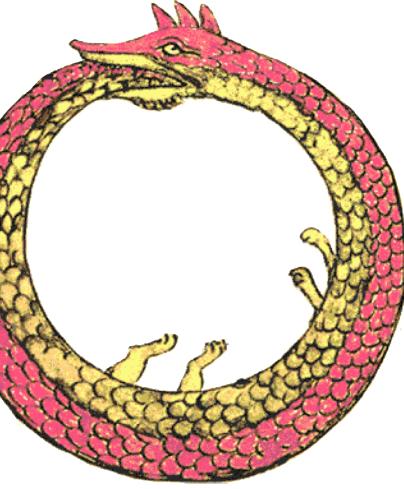
$$x^2 - x - 1 = 0$$

$$x = (1 \pm \sqrt{5}) / 2$$

- La suite de Fibonacci croît donc comme les puissances du nombre d'or

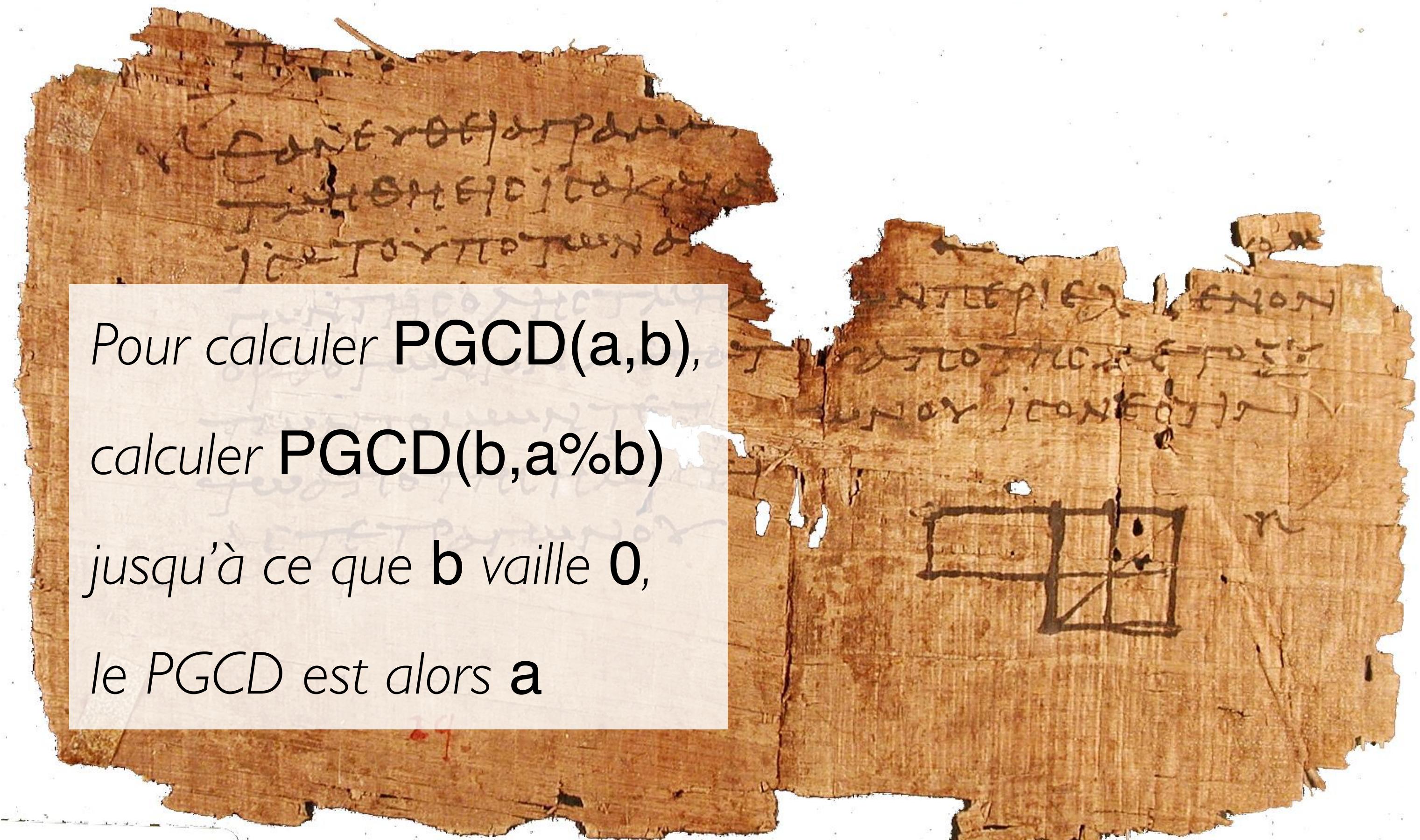
$$\phi = (1 + \sqrt{5}) / 2 = 1.618\dots$$



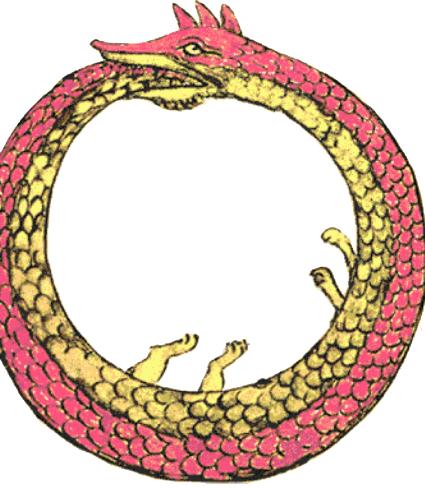


# Algorithme d'Euclide

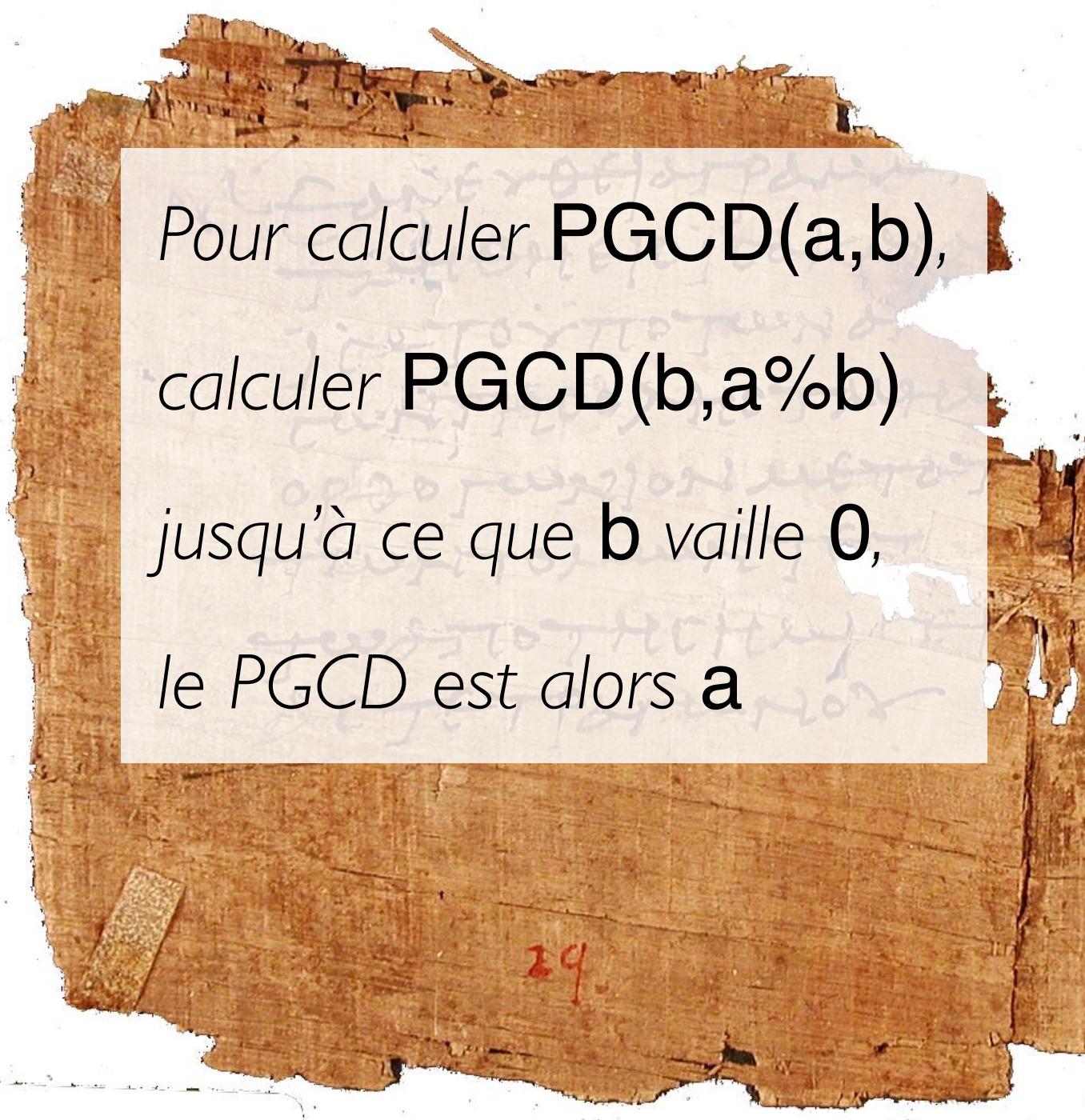
- Quel est le pire cas pour le calcul du PGCD ?



# Algorithme d'Euclide

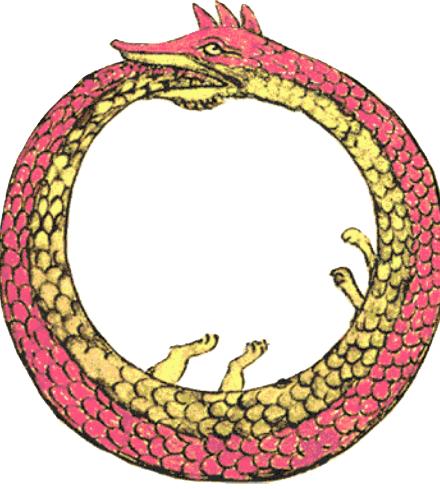


- De manière générale, pour  $a > b$ ,  
 $a \% b = a - n * b$ , avec  $n \geq 1$
- Dans le pire cas, n diminue le moins vite  
 $n=1$   
 $a \% b = a - b$
- La pire suite est donc  
 $(a,b) \rightarrow (b,a-b) \rightarrow (a-b, 2b-a) \rightarrow \dots$
- La suite de Fibonacci ...

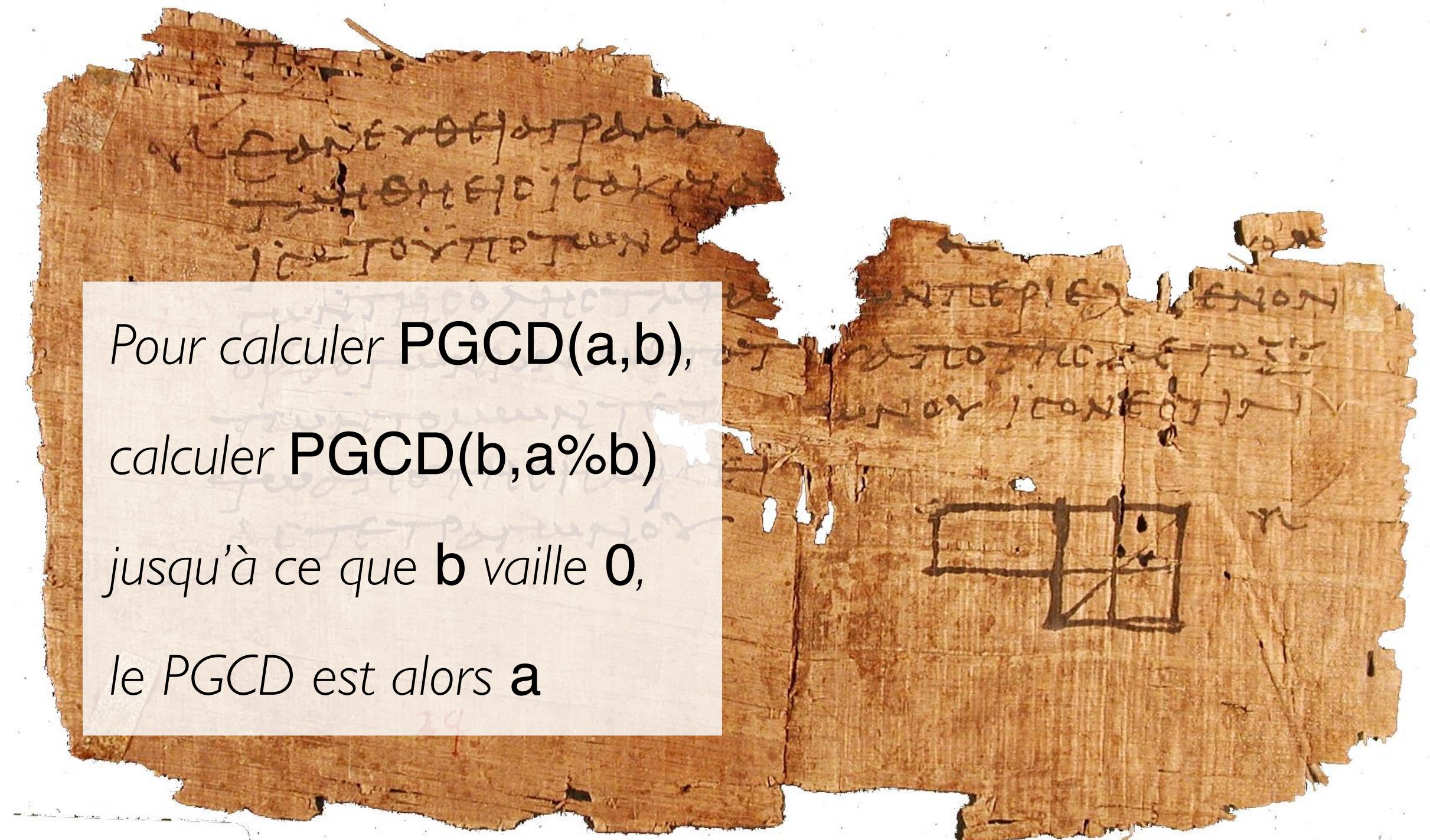


a	b
34	21
21	13
13	8
8	5
5	3
3	2
2	1
1	1
1	0

# Algorithme d'Euclide



- Il faut de l'ordre  $n$  étapes pour calculer  $\text{PGCD}(F(n), F(n-1))$
- ... au pire  $n$  étapes pour calculer  $\text{PGCD}(\phi^n, \dots)$
- ... au pire  $\log_\phi(n)$  étapes pour calculer  $\text{PGCD}(n, \dots)$
- L'algorithme d'Euclide a une complexité logarithmique dans le pire des cas



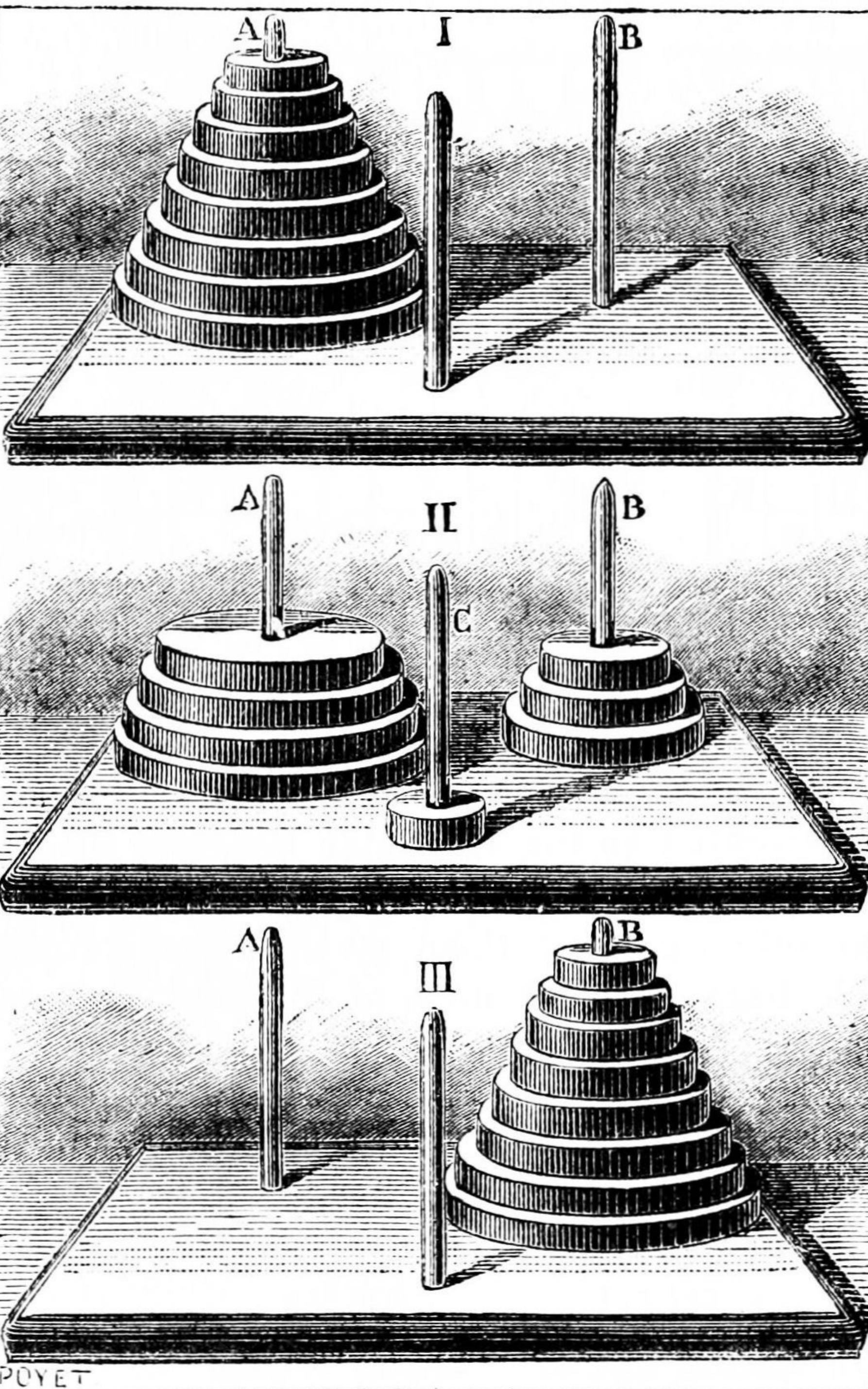
## 2.3. Tours de Hanoï

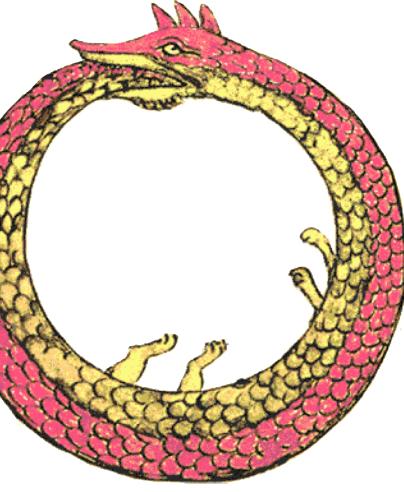


# Les tours de Hanoï

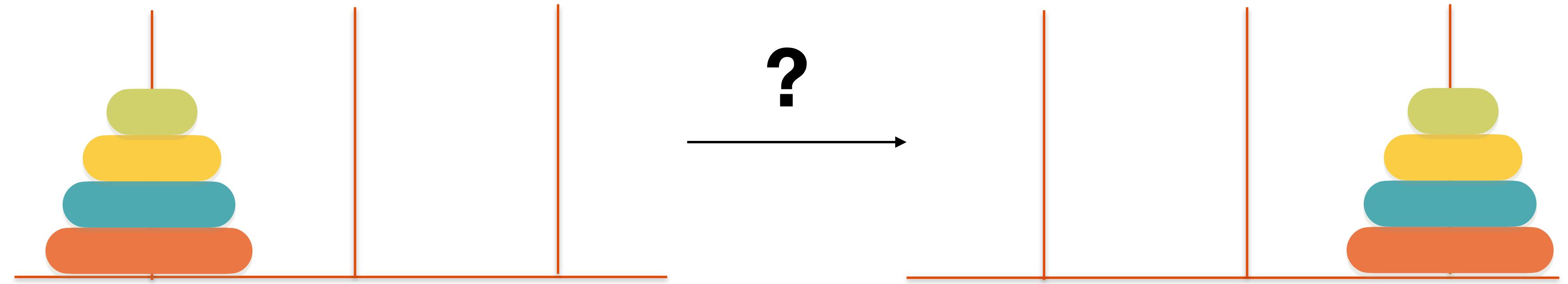
« ... dans le grand temple de Bénarès, au-dessous du dôme qui marque le centre du monde, trois aiguilles de diamant, plantées dans une dalle d'airain, hautes d'une coudée et grosses comme le corps d'une abeille.

Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet. C'est la tour sacrée du Brahmâ. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille sur la troisième, sans s'écartier des règles fixes que nous venons d'indiquer, et qui ont été imposées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes! »

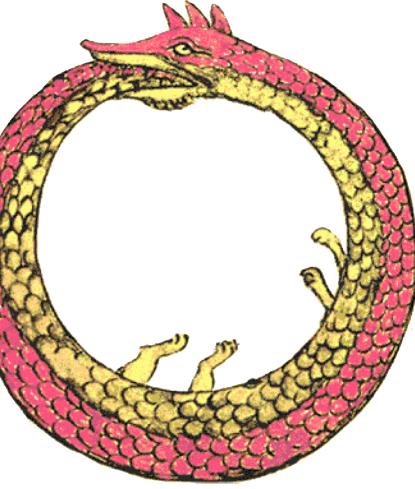




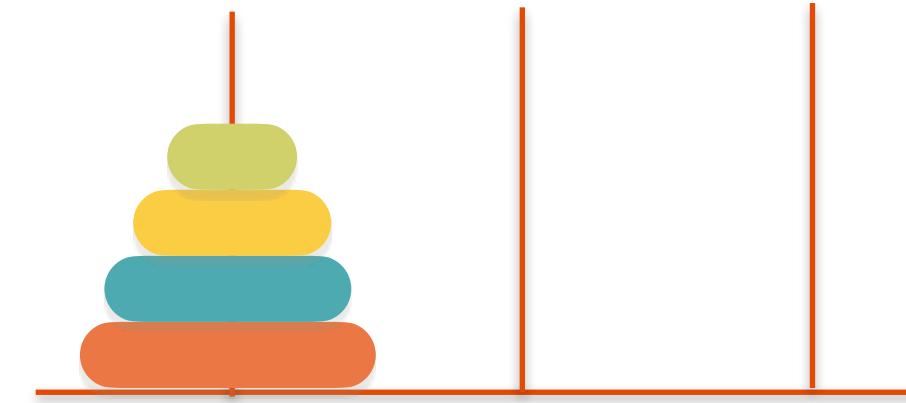
# Règles du jeu



- On déplace un seul disque à la fois.
- On ne peut pas placer un disque plus grand sur un plus petit

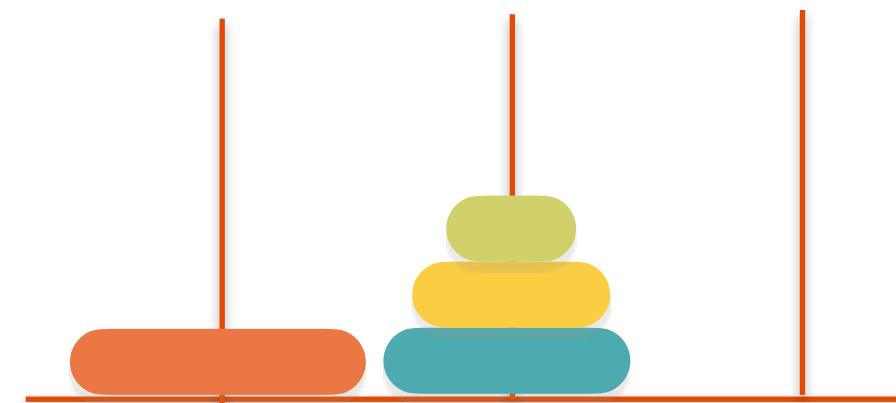


# Principe de la solution



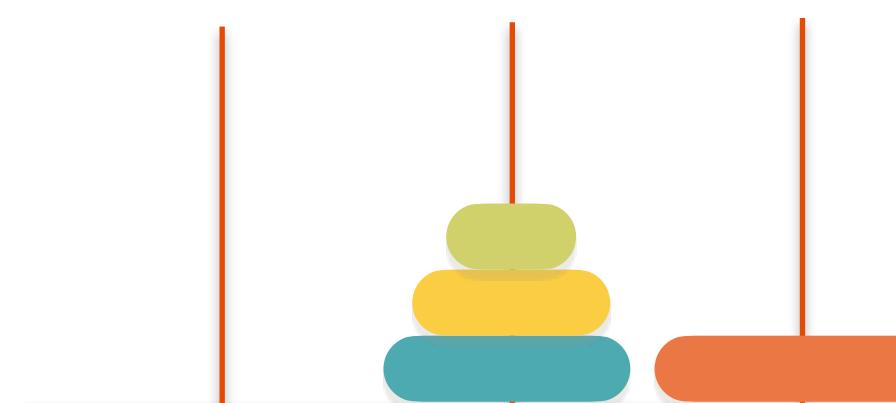
**Cas trivial**  
**Pas de disque à déplacer**

Pour déplacer n disques de gauche à droite

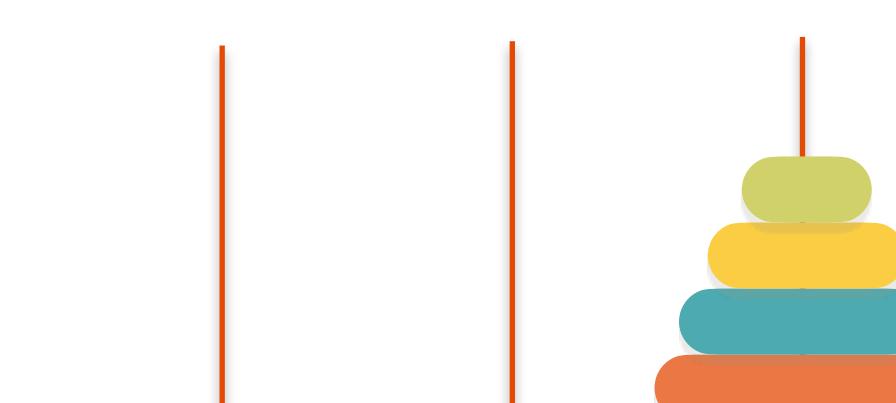


**Cas général**

Déplacer n-1 disques de gauche au milieu

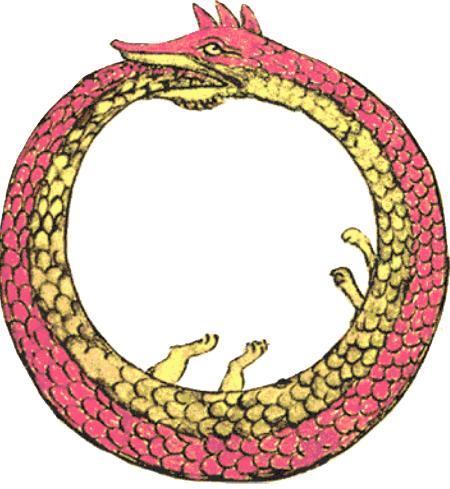


Déplacer le n<sup>ième</sup> disque de gauche à droite

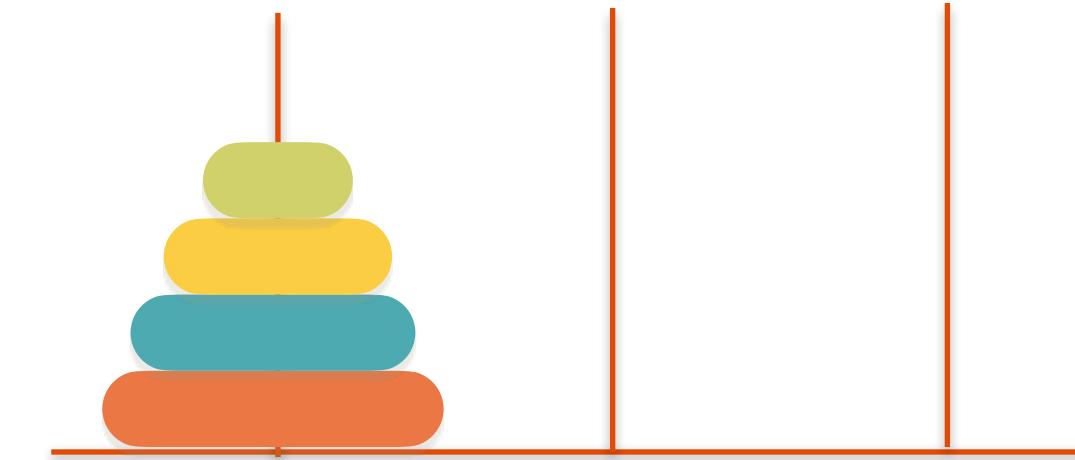


**Cas général**

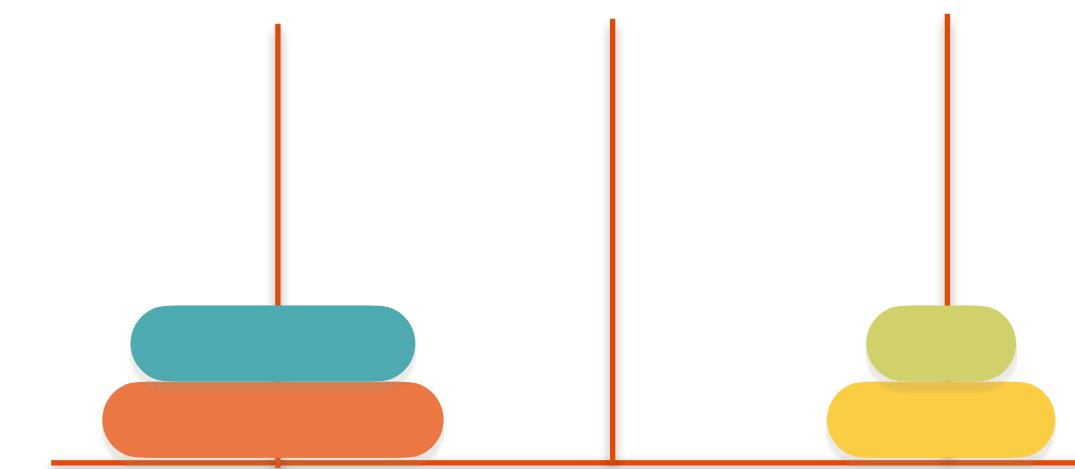
Déplacer n-1 disques du milieu à droite



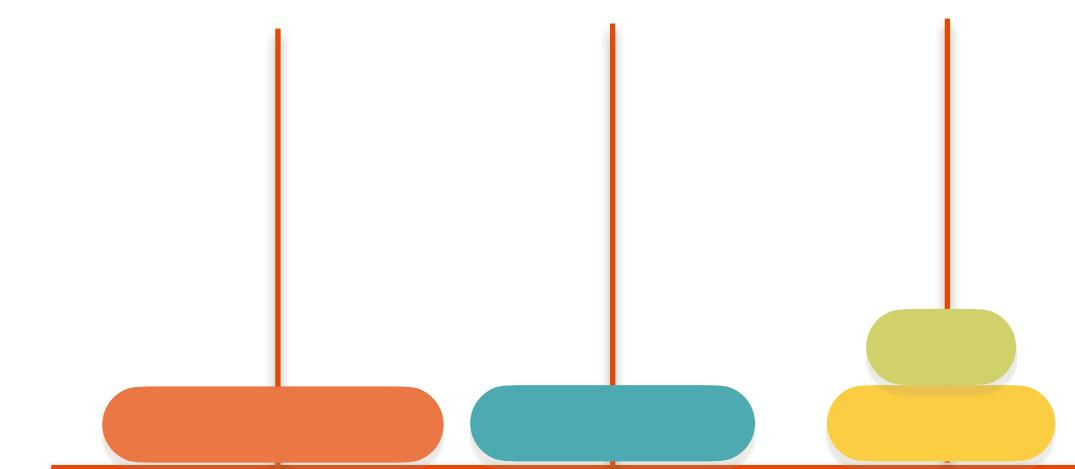
# Décomposons la première étape



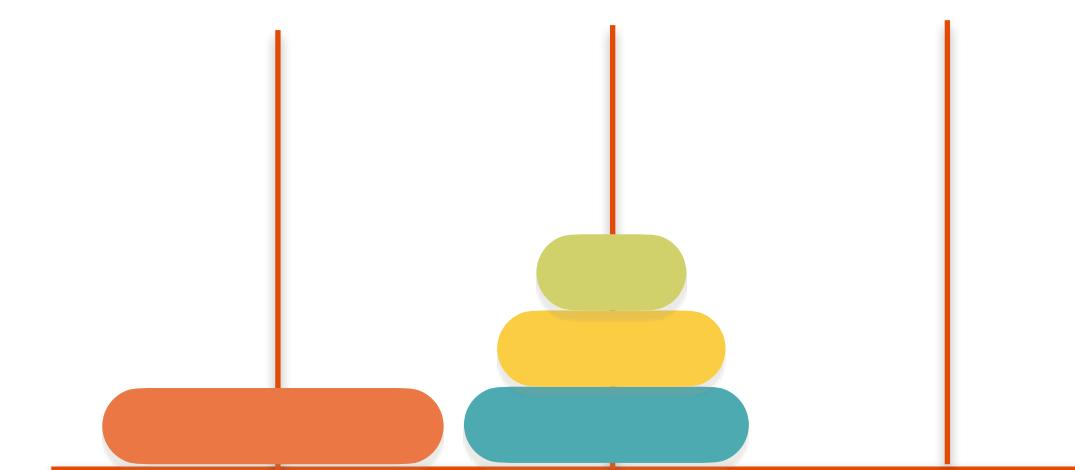
Pour déplacer  $n-1$  disques de gauche au milieu



Déplacer  $n-2$  disques de gauche à droite

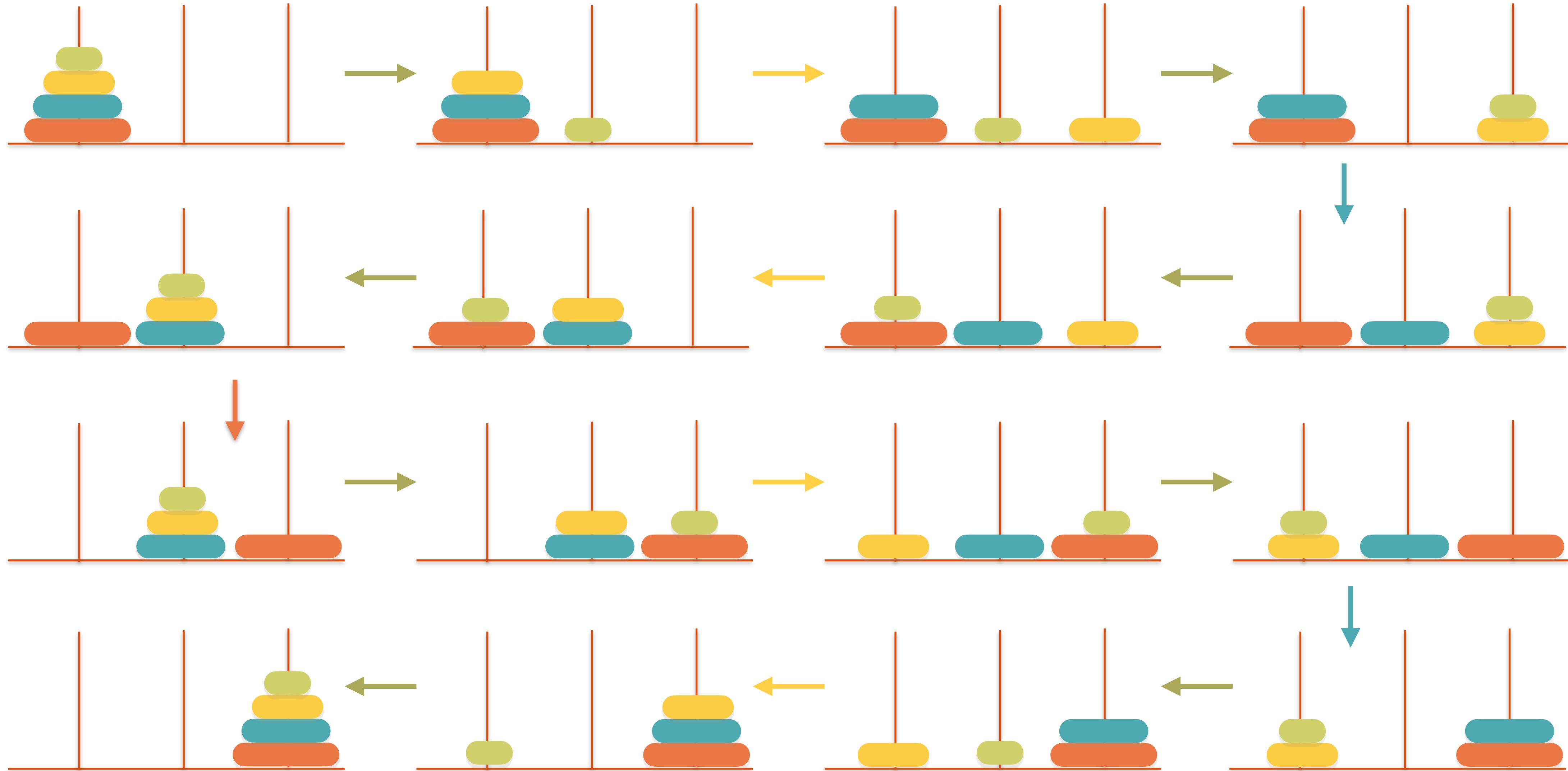
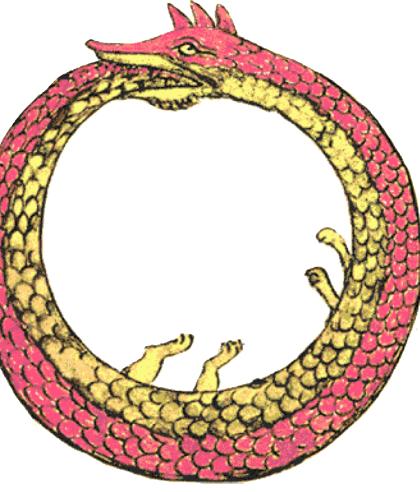


Déplacer le  $(n-1)^{\text{ième}}$  disque de gauche au milieu



Déplacer  $n-2$  disques de droite au milieu

## Toutes les étapes pour 4 disques



# CodeCheck it !

```
typedef vector<char> Tour;
```

```
vector<Tour> tours(3);
```

```
void transfert(Tour& from, Tour& via, Tour& to, int n)
```

```
{
```

```
    // A COMPLETER
```

```
    void transfert(Tour& from, Tour& via, Tour& to, int n)
```

```
{
```

```
    if(n == 0) return;
```

```
    transfert(from,to,via,n-1);
```

```
    to.push_back(from.back());
```

```
    from.pop_back();
```

```
    display();
```

```
    transfert(via,from,to,n-1);
```

```
    // transfère n disques de la tour from à la tour to
```

```
    // en utilisant la tour via comme intermédiaire
```

```
    // appeler display() après chaque mouvement de disque.
```

```
}
```

T0: A B C

T1:

T2:

T0: A B

T1:

T2: C

T0: A

T1: B

T2: C

T0: A

T1: B C

T2:

T0:

T1: B C

T2: A

T0: C

T1: B

T2: A

T0: C

T1:

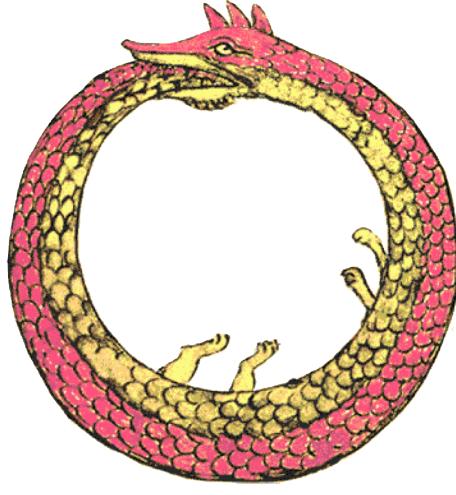
T2: A B

T0:

T1:

T2: A B C

# Algorithme et complexité



**Algorithme :**

Transférer n disques du piquet O (origine) vers le piquet D (destination) via le piquet I (intermédiaire):

**si**  $n > 0$  **alors**

*Transférer*  $n-1$  disques de O vers I via D

transférer le disque restant de O vers D

*Transférer*  $n-1$  disques de I vers D via O

**fin si**

**Complexité :**

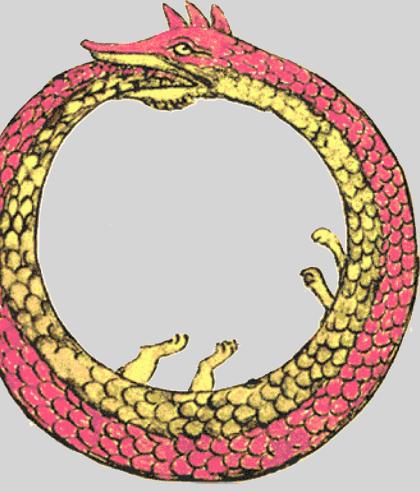
Soit  $T(n)$  le nombre de transferts pour  $n$  disques,

$$T(1) = 1$$

$$T(2) = 1 + T(1) + T(1)$$

$$T(n) = 1 + 2 T(n-1)$$

$$T(n) = 2^n - 1$$



# Exercice S-2.2

- Quelles sont les complexités des fonctions suivantes ?

```
int f1(unsigned n)
{
    if(n == 0)
        return 1;

    return f1(n-1) +
           f1(n-1) +
           f1(n-1);
}
```

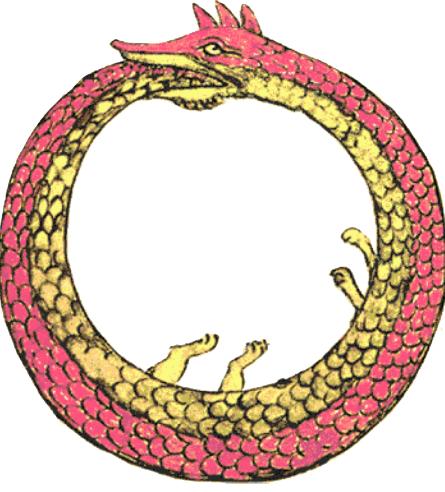
```
int f2(unsigned n)
{
    if(n == 0)
        return 1;

    return f2(n/2) +
           f2(n/2);
}
```

## 2.4. Permutations



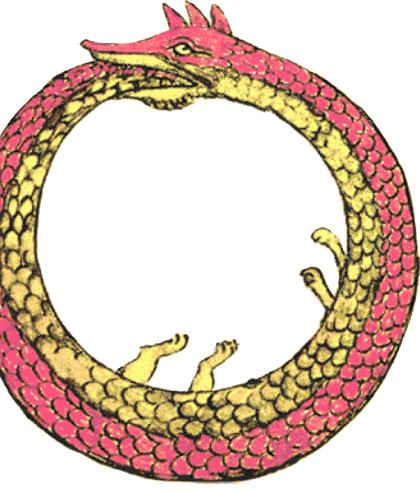
# Scrabble



Pour faire jouer un programme au scrabble, il faut ...

- Un dictionnaire
- Un algorithme de recherche efficace dans ce dictionnaire : dichotomique en  $O(\log(n))$
- Générer toutes les **permutations** possibles de nos lettres





# Approche récursive

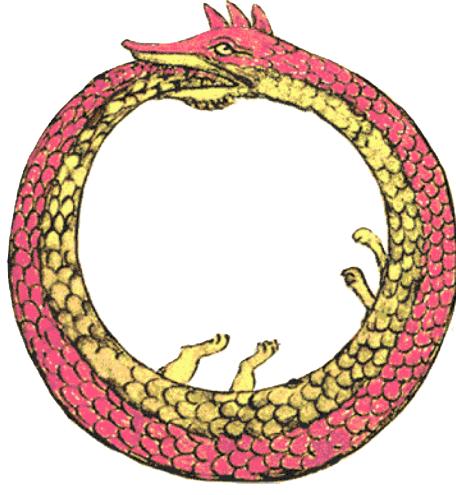
- Générer toutes les **permutations des n premiers caractères** d'une chaîne de caractères S

**Entrée:** S, par exemple ABCD  
n, par exemple 3

**Sortie:** toutes les permutations des n premiers caractères de S, par exemple  
{ ABCD, BACD, ACBD, CABD, BCAD, CBAD }

- **Cas général** : placer chacun des caractères en dernière position, générer toutes les permutations des n-1 caractères restants aux n-1 positions restantes
- **Cas trivial** : 1 seule permutation pour une chaîne de 1 caractère

# Permuter les n premiers caractères



## Entrée:

S, par exemple ABCD  
n, par exemple 3

## Sortie:

toutes les permutations des n premiers caractères de S, par exemple { ABCD, BACD, ACBD, CABD, BCAD, CBAD }

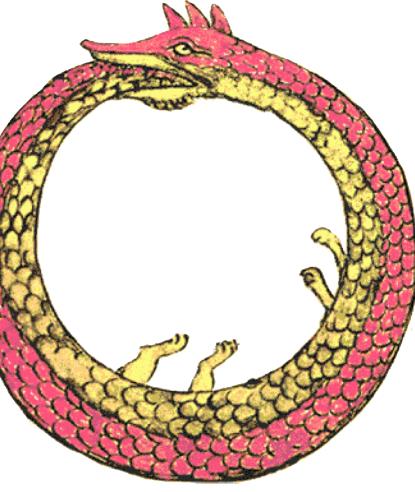
## Algorithme:

```
fonction permuter(S,n)
    si n vaut 1
        S est la seule permutation
    sinon
        pour toutes les lettres c de S, boucler
            Placer c en dernière position
            permuter(S,n-1)
        fin boucler
    fin si
```

Paramètre de récursivité: n

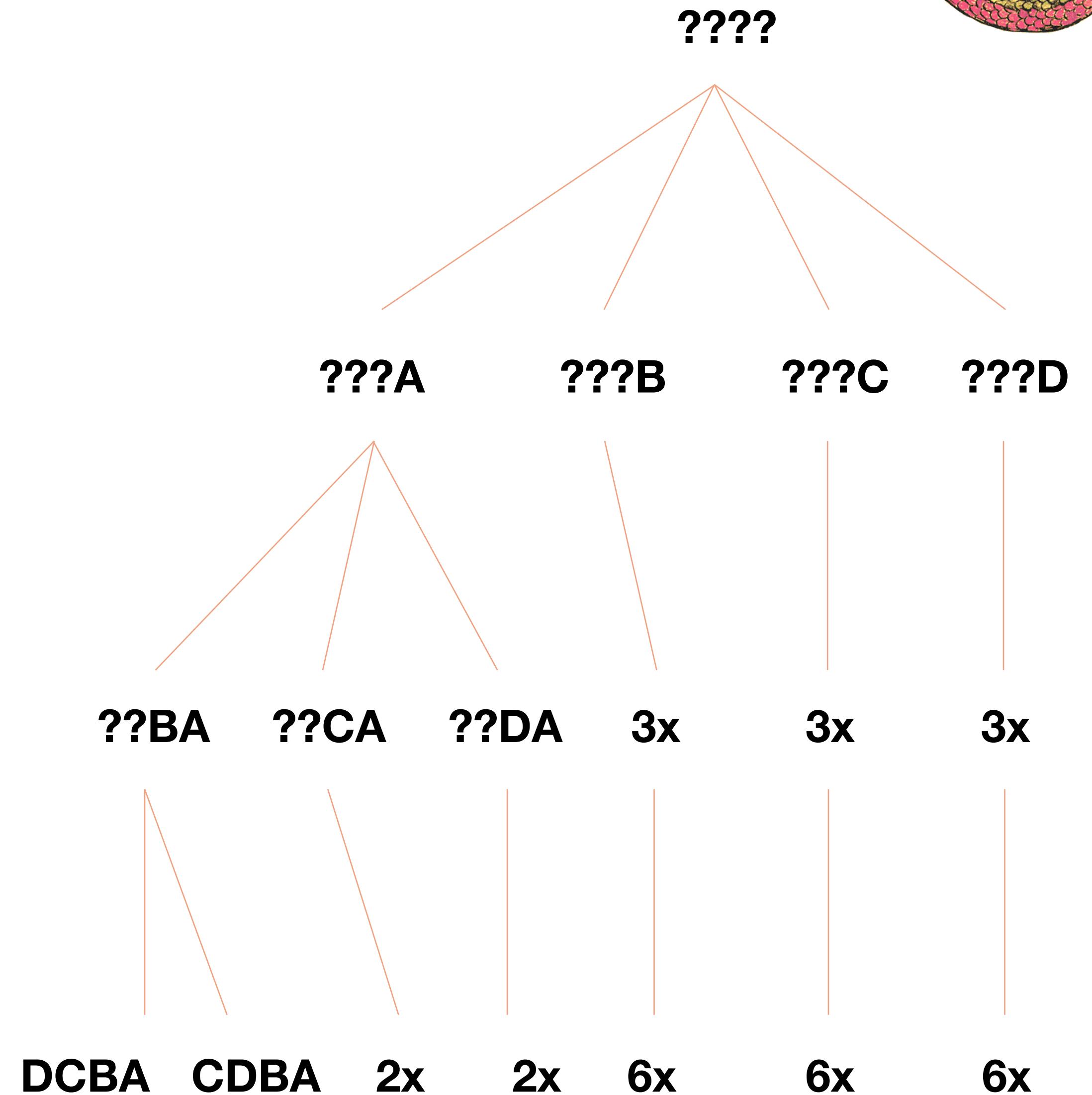
Cas trivial

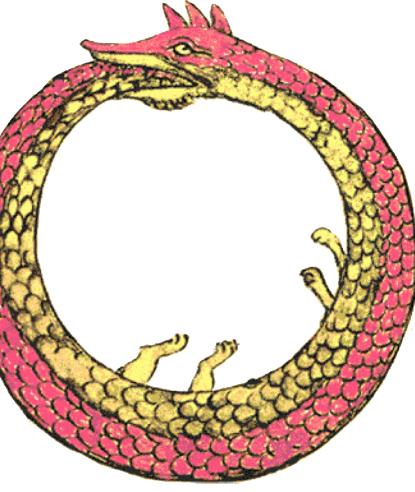
Cas général



# Complexité ?

- Il y a  $n!$  permutations à générer
- L'algorithme sera donc au minimum  $O(n!)$
- Combien d'appels récursifs ?
- Combien d'écritures / d'échange de caractères ?





# Nombre d'appels récursifs

$$n! \sum_{k=1}^n \frac{1}{k!} \approx (e-1) \cdot n! \approx 1.71 \cdot n! = O(n!)$$

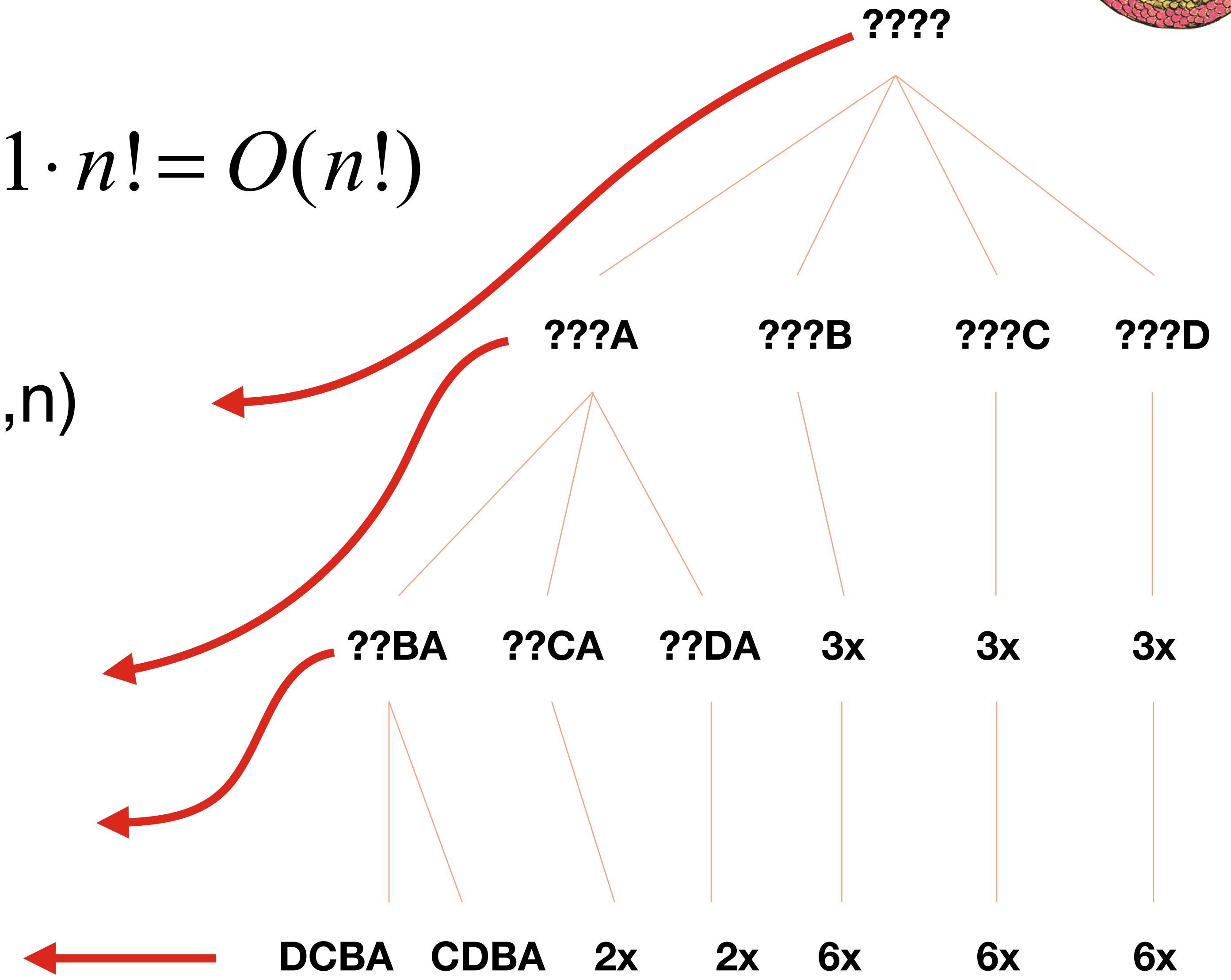
$n!/n! = 1$  appel à `permute(S,n)`

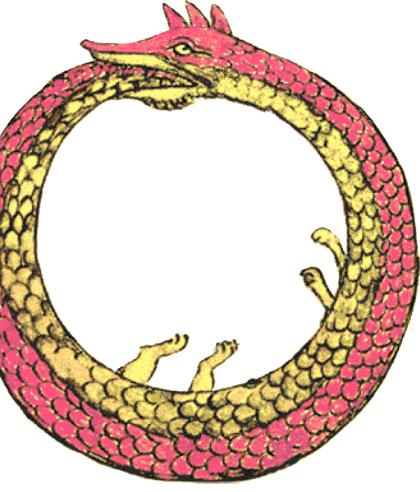
$n!/k!$  appels à `permute(S,k)`

$n!/6$  appels à `permute(S,3)`

$n!/2$  appels à `permute(S,2)`

$n!$  appels au cas trivial





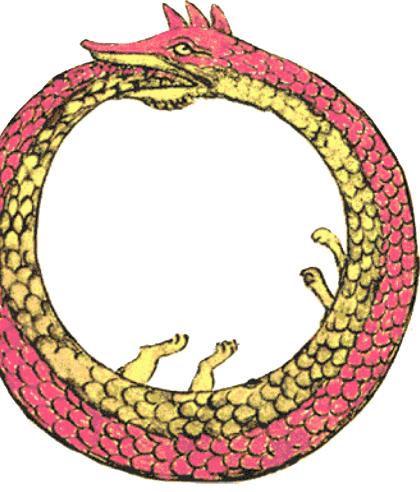
# Boucler ? Placer ?

Comment ...

- ... boucler sur les n premières lettres de S ?
- ... placer c en dernière position ?

**Algorithme:**

```
fonction permuter(S,n)
  si n vaut 1
    S est la seule permutation
  sinon
    pour toutes les lettres c de S, boucler
      Placer c en dernière position
      permuter(S,n-1)
    fin boucler
  fin si
```



# Approche naïve

```
fonction permuter(S,n)
    si n vaut 1
        traiter(S)
    sinon
        pour i allant de 1 à n, boucler
            permuter(S,n-1)
            échanger S(i) et S(n)
        fin boucler
    fin si
```

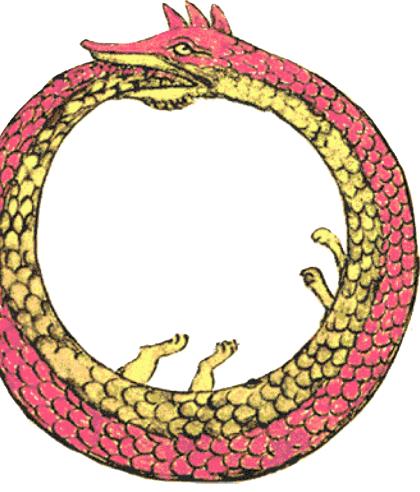
Comment passer le paramètre S de la fonction ?

- Par **valeur** ? Complexité  $O(n \cdot n!)$
- Par **référence** ? L'algorithme ne fonctionne pas

La boucle « **pour i allant de 1 à n** » n'est pas équivalente à « **pour toutes les lettres c de S** » si l'ordre des caractères de S change en cours de route

ABC  
BAC  
CBA  
BCA  
CAB  
ACB

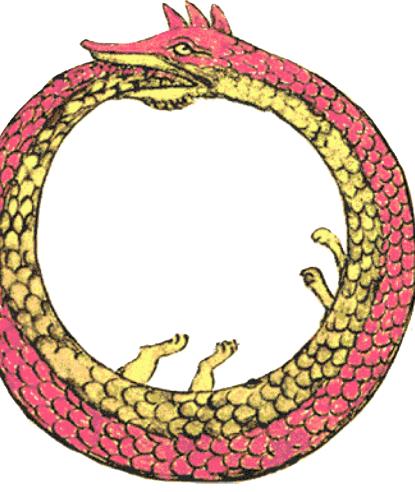
ABC  
BAC  
CAB  
ACB  
ABC  
BAC



# Double échange

```
fonction permute(S,n)
  si n vaut 1
    traiter(S)
  sinon
    pour i allant de 1 à n, boucler
      échanger S(i) et S(n)
      permute(S,n-1)
      échanger S(i) et S(n)
    fin boucler
  fin si
```

- Pour que la boucle pour i allant de 1 à n soit équivalente à pour toutes les lettres c de S,
- On remet la lettre S(i) à sa place avant de passer à l'itération suivante
- 2 échanges par appel récursif, donc en tout  $3.43 n!$



# Sans échange inutile

- Echanger  $S(i)$  et  $S(n)$  est inutile pour  $i=n$
- $2(k-1)$  échanges dans  $\text{permute}(S,k)$

**fonction** permute( $S,n$ )

**si**  $n$  vaut 1  
  traiter( $S$ )

**sinon**

**pour**  $i$  allant de 1 à  $n-1$ , **boucler**

  échanger  $S(i)$  et  $S(n)$   
  permute( $S,n-1$ )

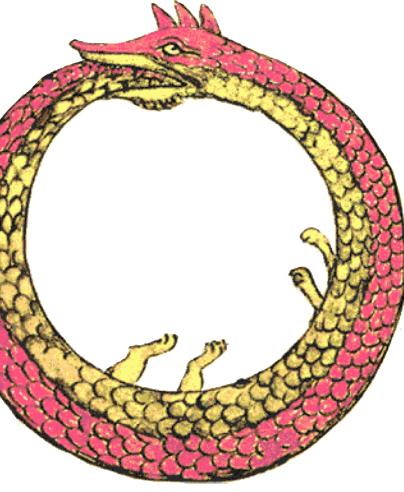
  échanger  $S(i)$  et  $S(n)$

**fin boucler**

permute( $S,n-1$ )

**fin si**

$$\begin{aligned} \text{Echanges}(n) &= 2 \sum_{k=2}^n (k-1) \frac{n!}{k!} \\ &= 2 \left( \sum_{k=2}^n \frac{k \cdot n!}{k!} - \sum_{k=2}^n \frac{n!}{k!} \right) \\ &= 2 \left( \sum_{k=2}^n \frac{n!}{(k-1)!} - \sum_{k=2}^n \frac{n!}{k!} \right) \\ &= 2 \left( \sum_{k=1}^{n-1} \frac{n!}{k!} - \sum_{k=2}^n \frac{n!}{k!} \right) \\ &= 2 \left( \frac{n!}{1!} - \frac{n!}{n!} \right) = 2(n!-1) \end{aligned}$$



# Algorithme de Heap

```
fonction permuter(S,n)
    si n vaut 1
        traiter(S)
    sinon
        permuter(S,n-1)
        pour i allant de 1 à n-1, boucler
            si n est pair
                échanger S(i) et S(n)
            sinon
                échanger S(1) et S(n)
            fin si
        permuter(S,n-1)
    fin boucler
fin si
```

- Permutations by interchanges.  
*B. R. Heap*, The Computer Journal, 6(3) (1963), pp. 293-298
- Nombre d'échanges optimal:  $n! - 1$