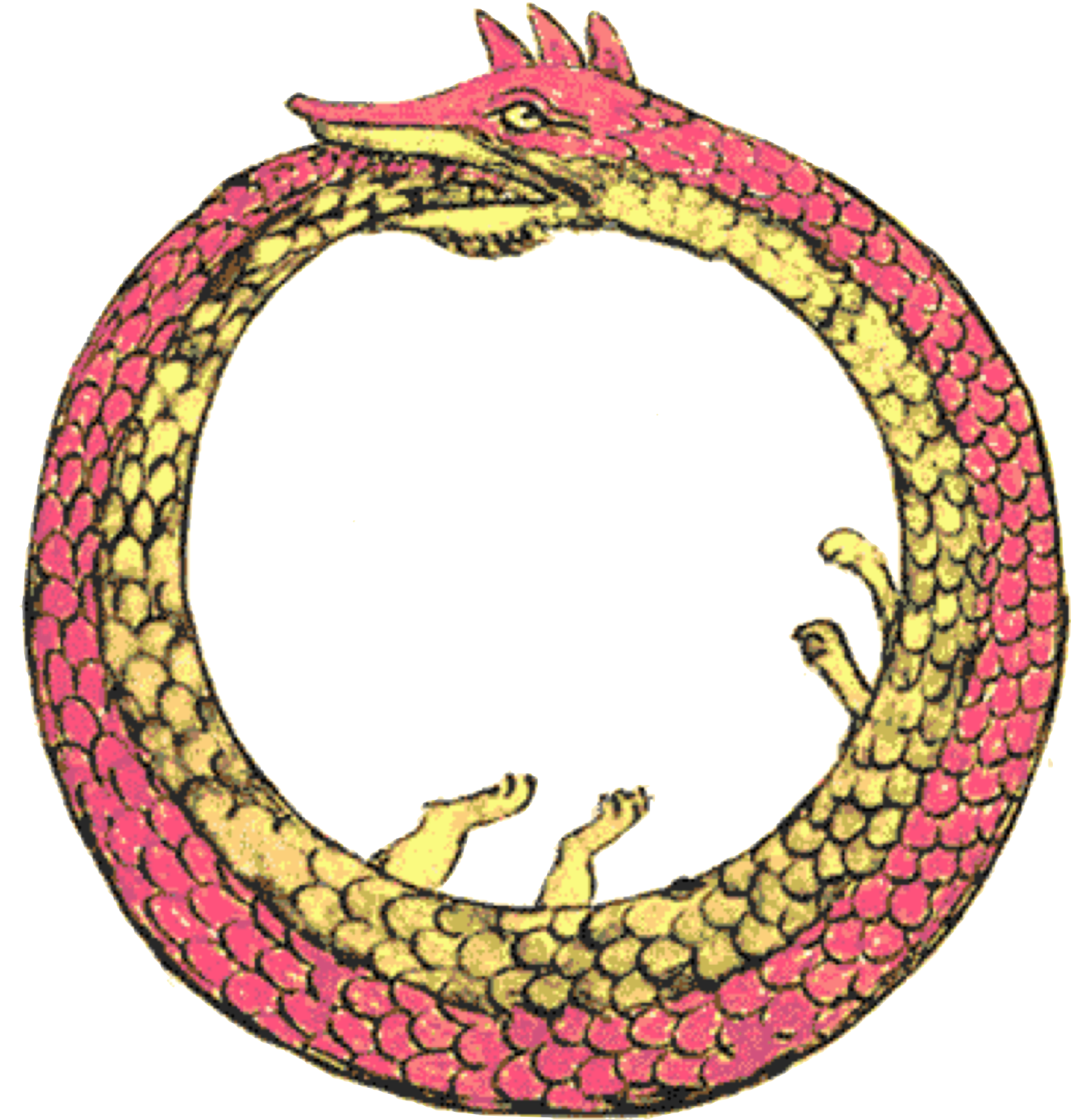
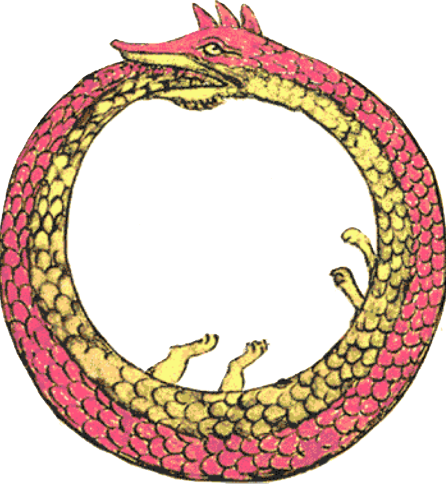


Récurtivité

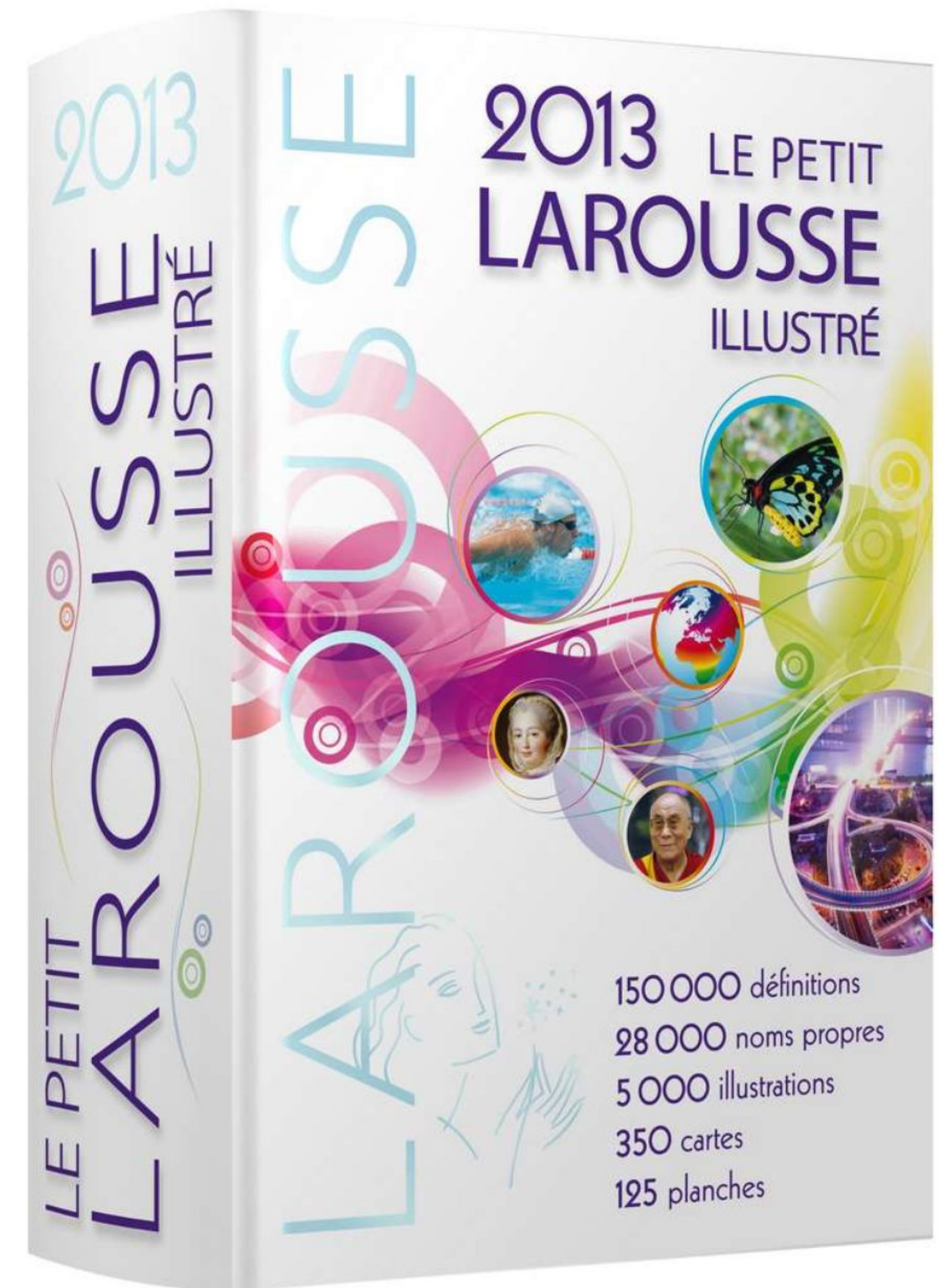


Définition

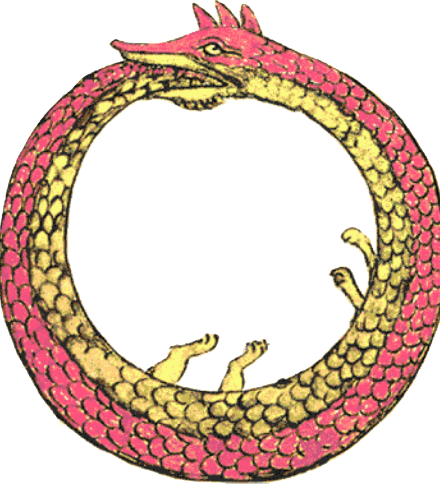


Récuratif, récurative, adjectif, (anglais recursive, du latin recursum, de recurrere, revenir en arrière)

- 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

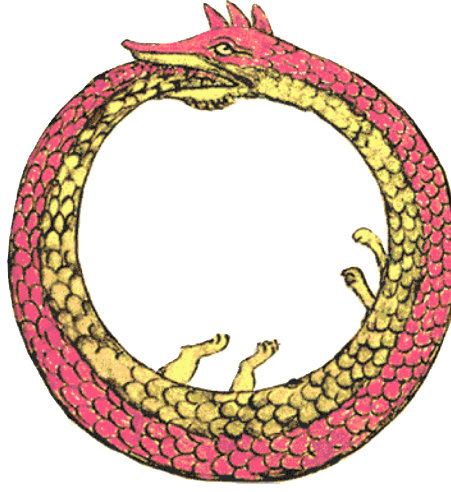


Exemple : Euclide

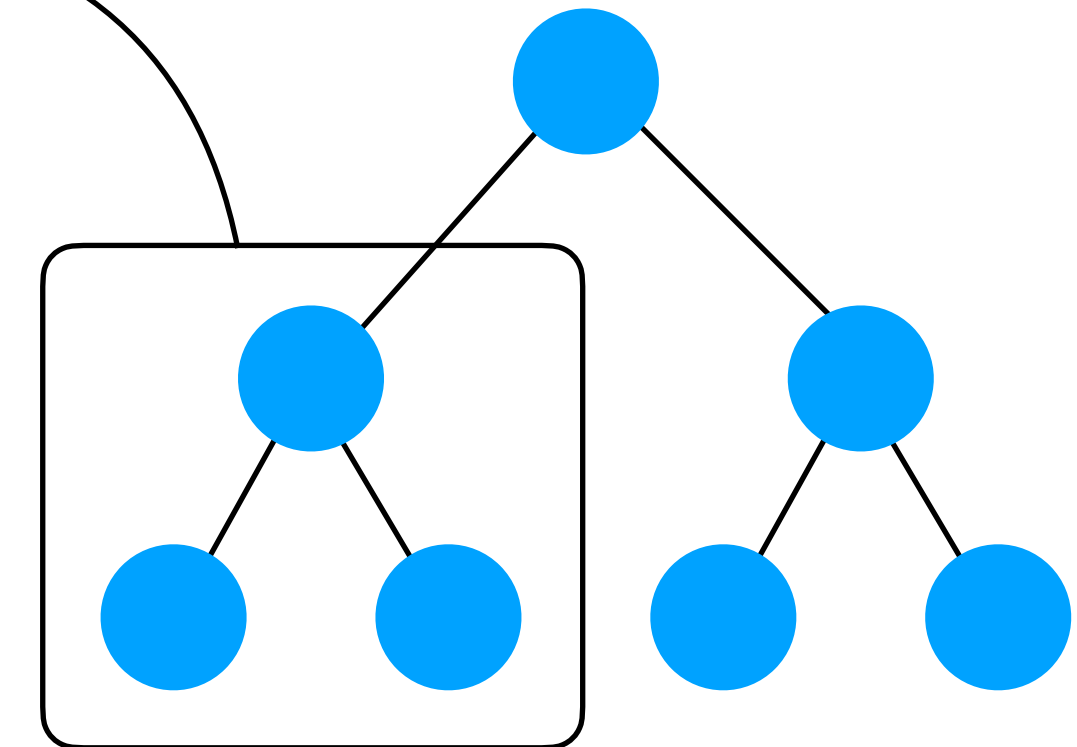
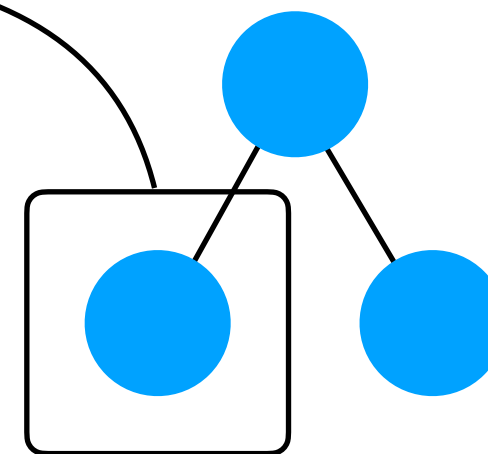
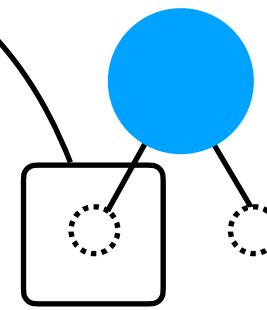


Pour calculer $\text{PGCD}(a, b)$,
calculer $\text{PGCD}(b, a \% b)$
jusqu'à ce que b vaille 0 ,
le PGCD est alors a

Structure réursive



Arbre vide

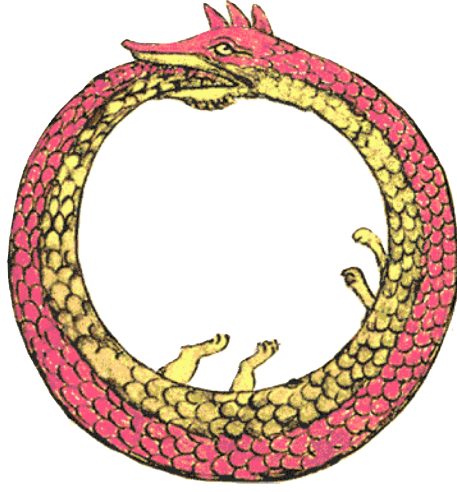


Un **arbre binaire** est

- soit un arbre vide, *Cas trivial*
- soit une structure composée d'une racine et de deux enfants qui sont eux-même des **arbres binaires**. *Cas général*
utilisant la récursivité

2.1. Factorielle

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



Factorielle

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

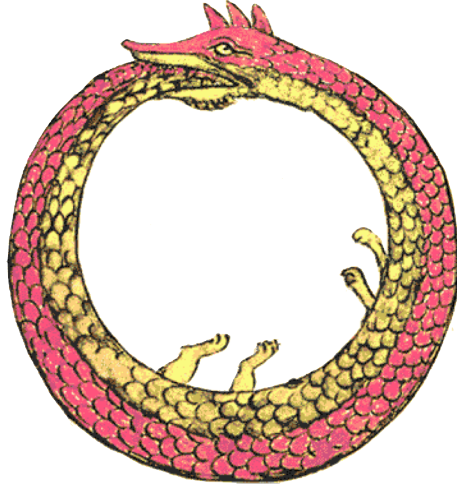
$$n! = \begin{cases} \underline{1} & \text{si } n = 0 \text{ ou } 1 \\ n \times \underline{(n-1)!} & \text{sinon} \end{cases}$$

Cas trivial

Cas général
utilisant la récursivité

```
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
```

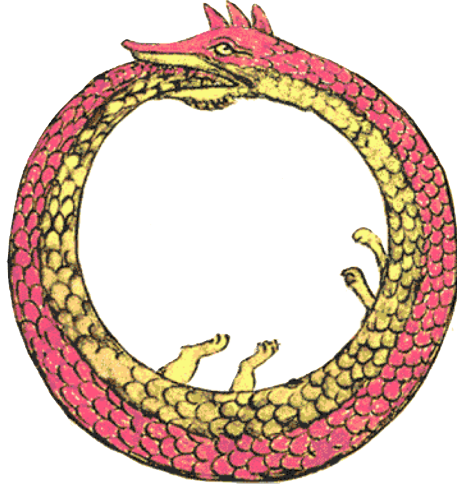
Calculer 4!

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!

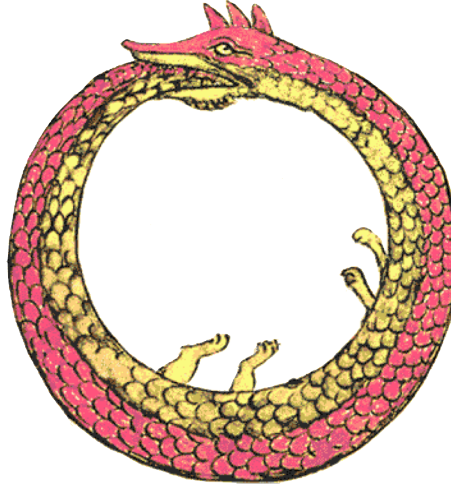
3 != 0

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

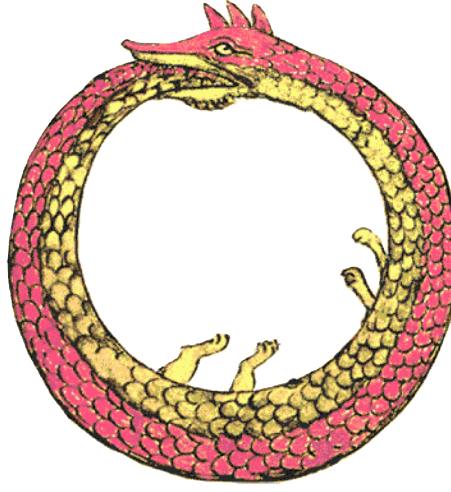
Calculer 1!

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!

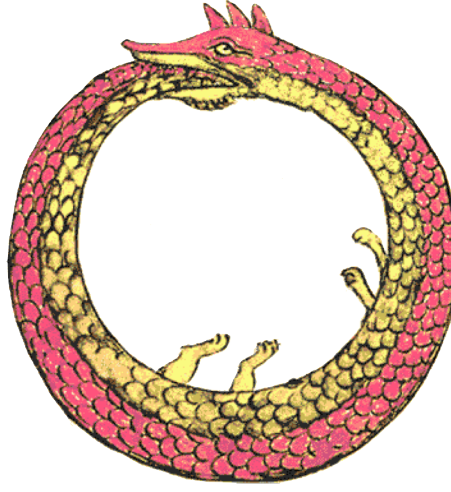
retourner 1 x 0!

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

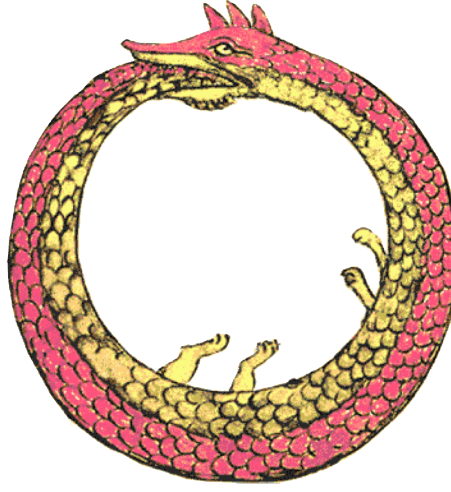
retourner 1 x 0!

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
```

Calculer 4!

4 != 0

Calculer 3!

3 != 0

Calculer 2!

2 != 0

Calculer 1!

1 != 0

Calculer 0!

0 == 0

retourner 1

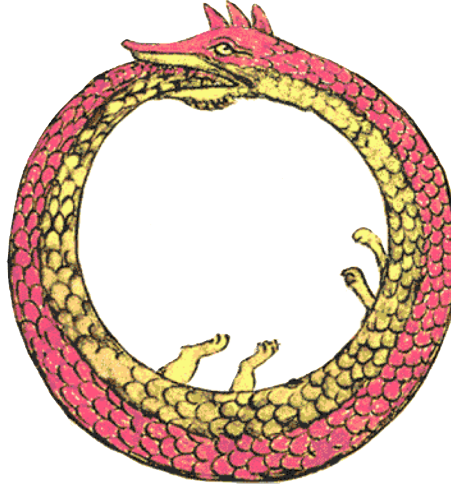
retourner 1 x 0! = 1

retourner 2 x 1! = 2

retourner 3 x 2! = 6

retourner 4 x 3! = 24

Cas trivial, cas général



```
fonction factorielle(n)
```

```
  si n vaut 0 alors  
    retourner 1
```

```
  sinon  
    retourner n x factorielle(n-1)  
  fin si
```

Paramètre de
récursivité

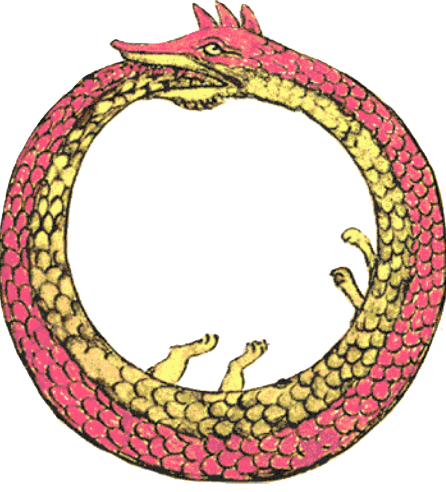
Cas trivial

Fin de la récursion

Cas général

Nouvel appel récursif

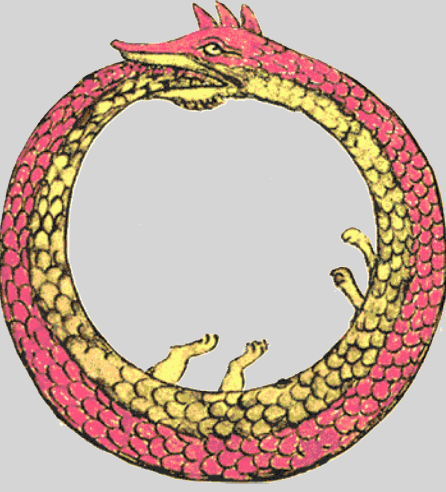
Complexité ?



- 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, $O(n)$



- Quelles sont les complexités de ces fonctions récursives ?

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

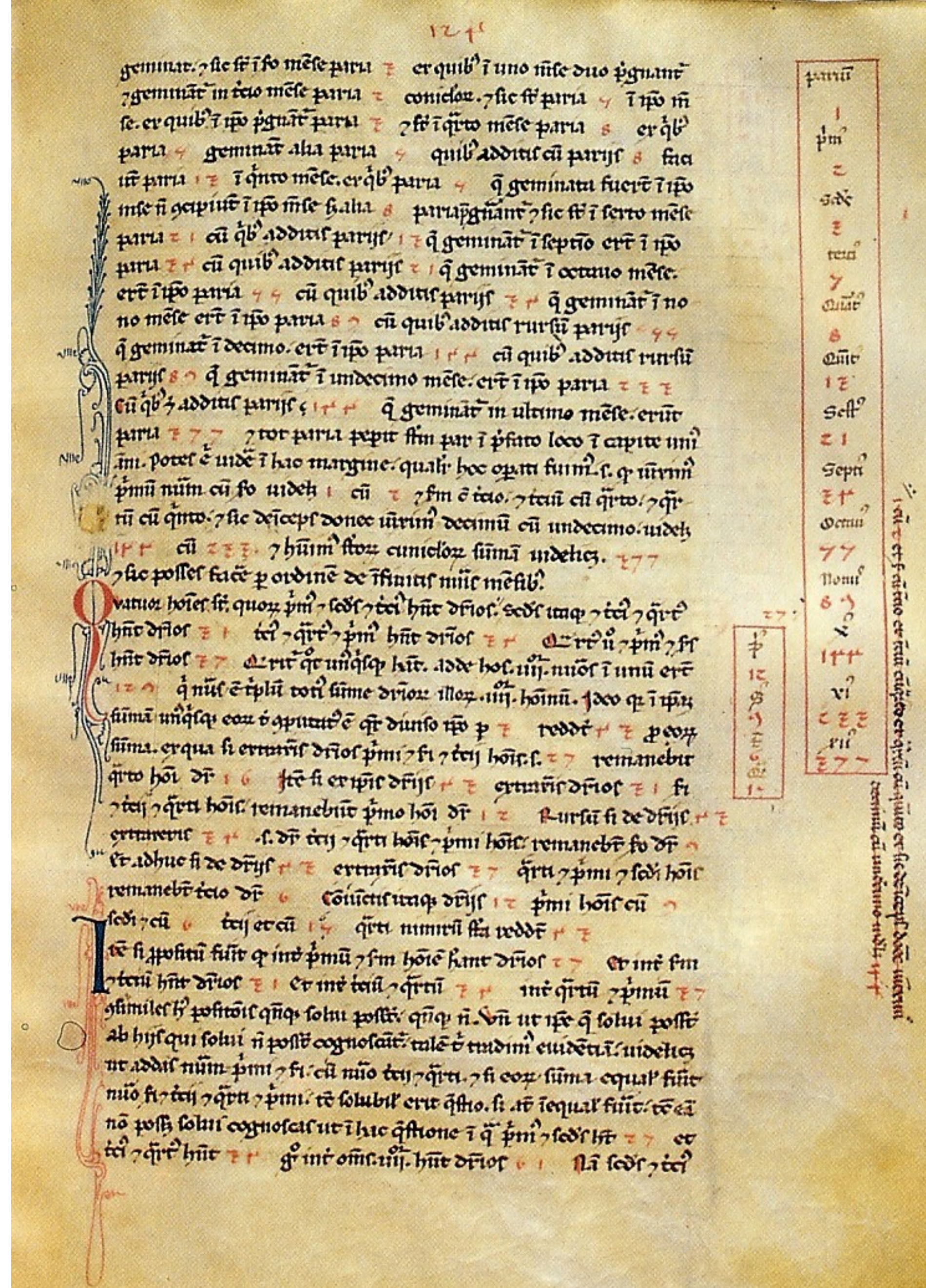
```
int f2(double n) {  
    if(n <= 1)  
        return 1;  
    else  
        return 1 + f2(n-2*sqrt(n)+1);  
}
```

2.2. Fibonacci

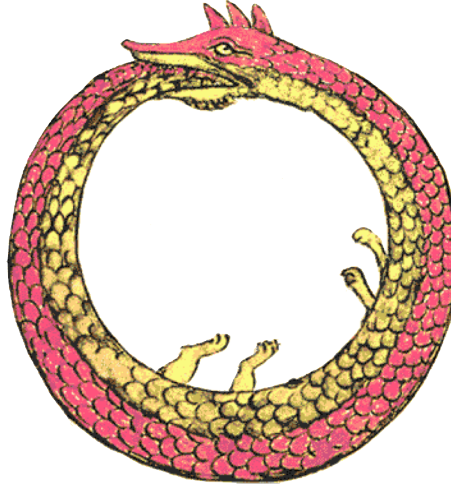


HEIG^{VD}

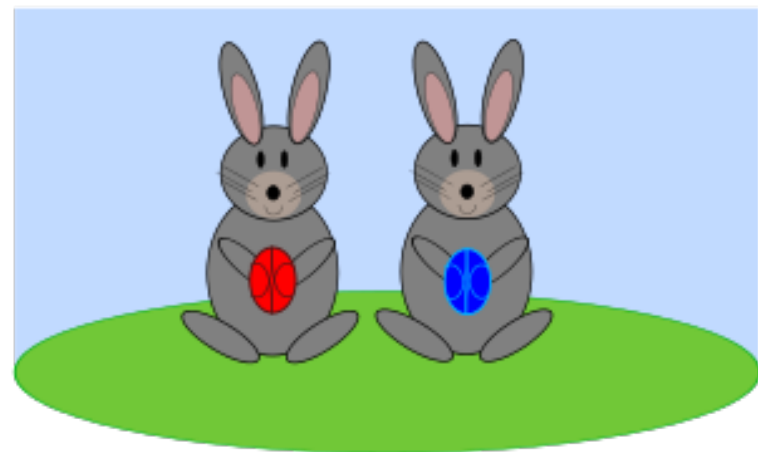
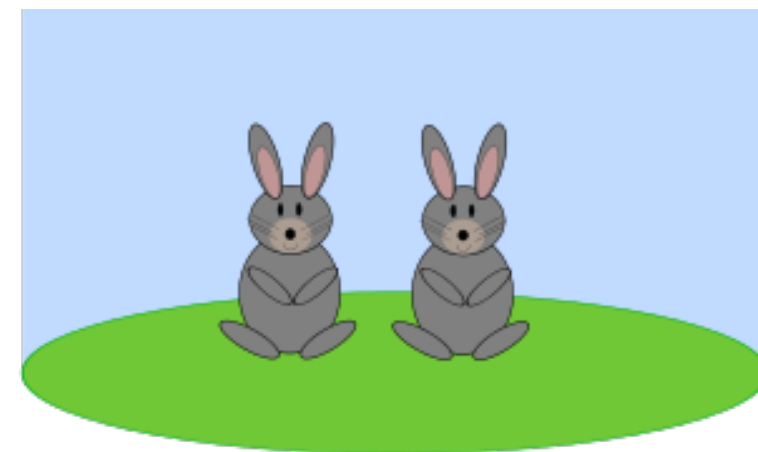
- Il s'intéresse en 1202 à la croissance d'une population de lapins idéalisée.
- On commence avec un couple de lapins nouveaux nés.
- A partir de l'âge de 2 mois, un couple de lapin engendre un couple de lapins tous les mois.
- Combien de couples de lapins obtient-on après n mois? (les lapins sont immortels)



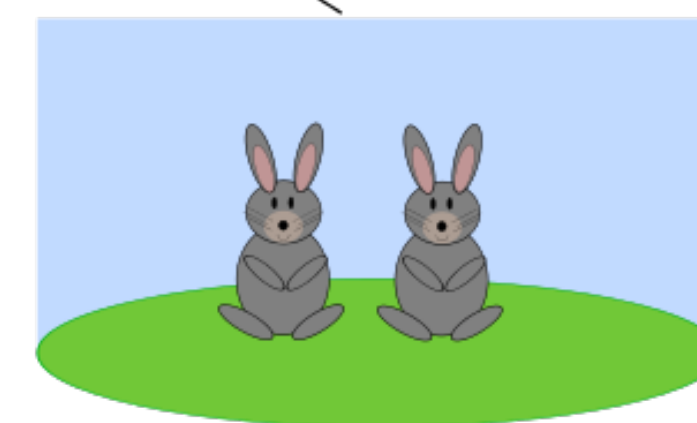
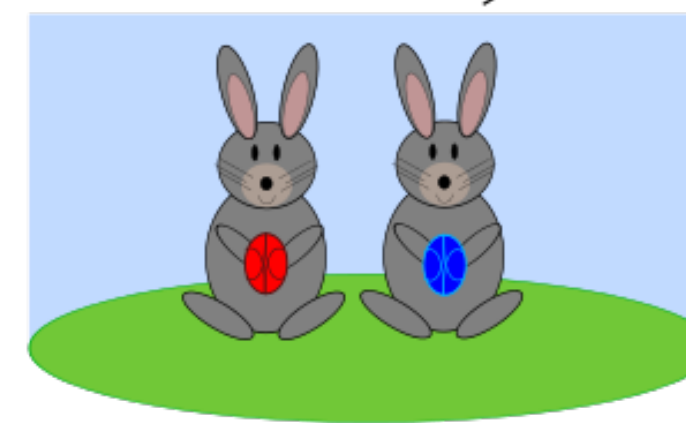
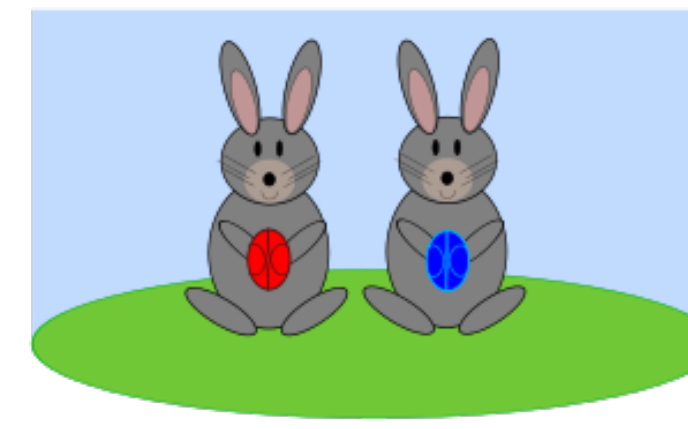
Graphiquement



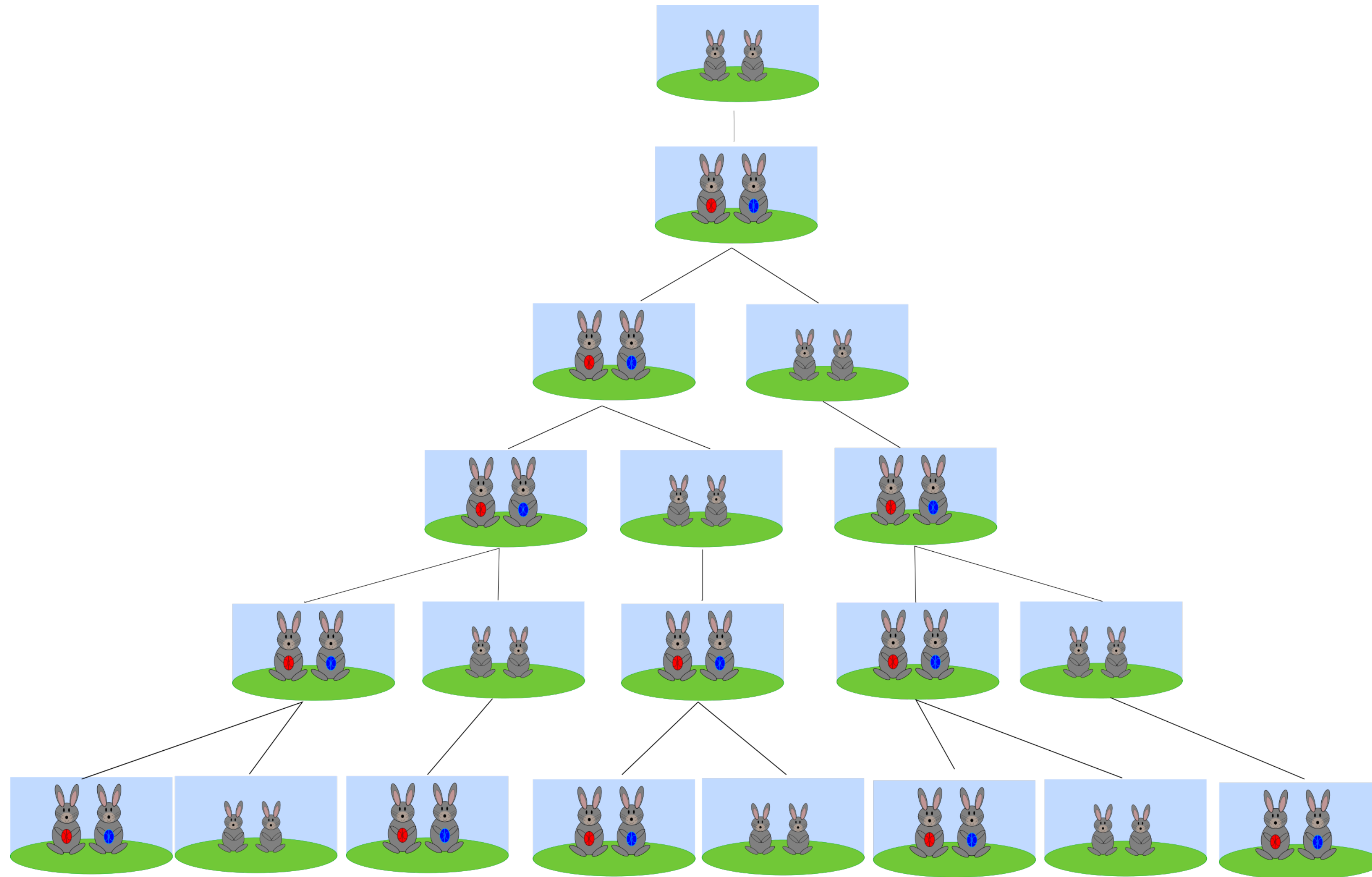
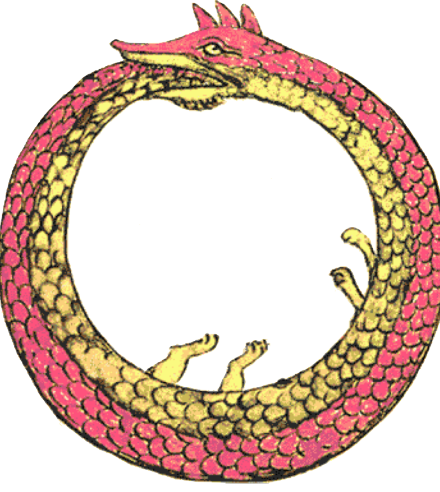
Les enfants
deviennent adultes
le mois suivant



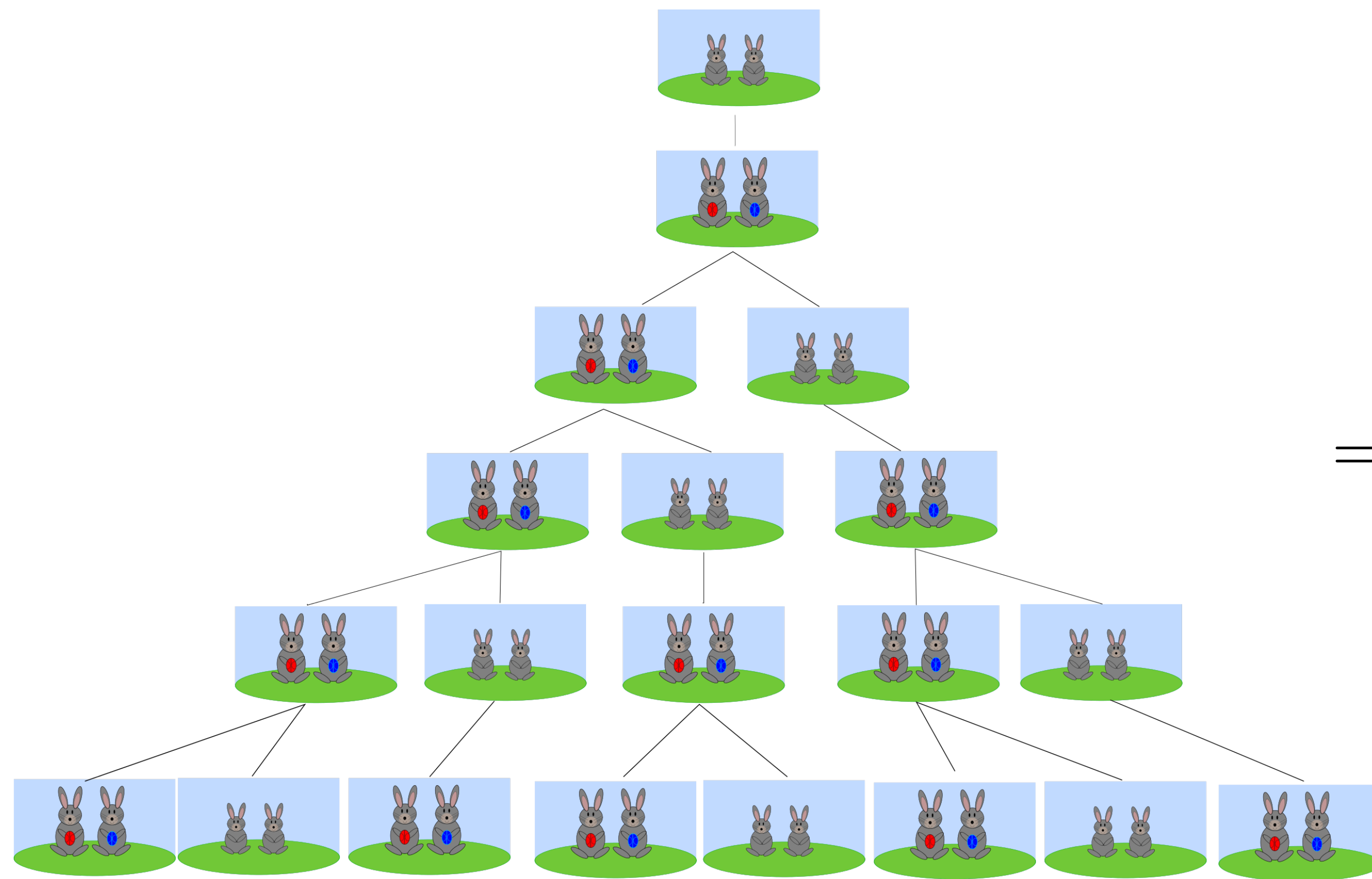
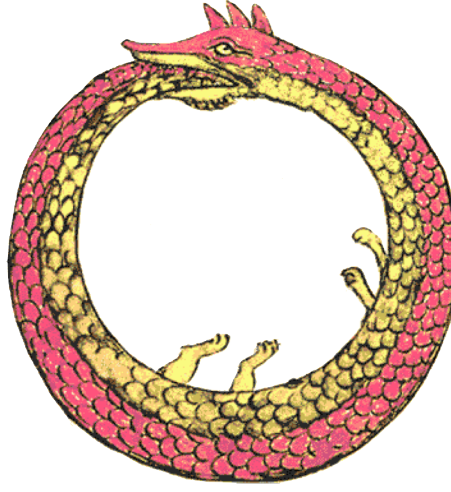
Les adultes font des
enfants, et survivent
tous les mois



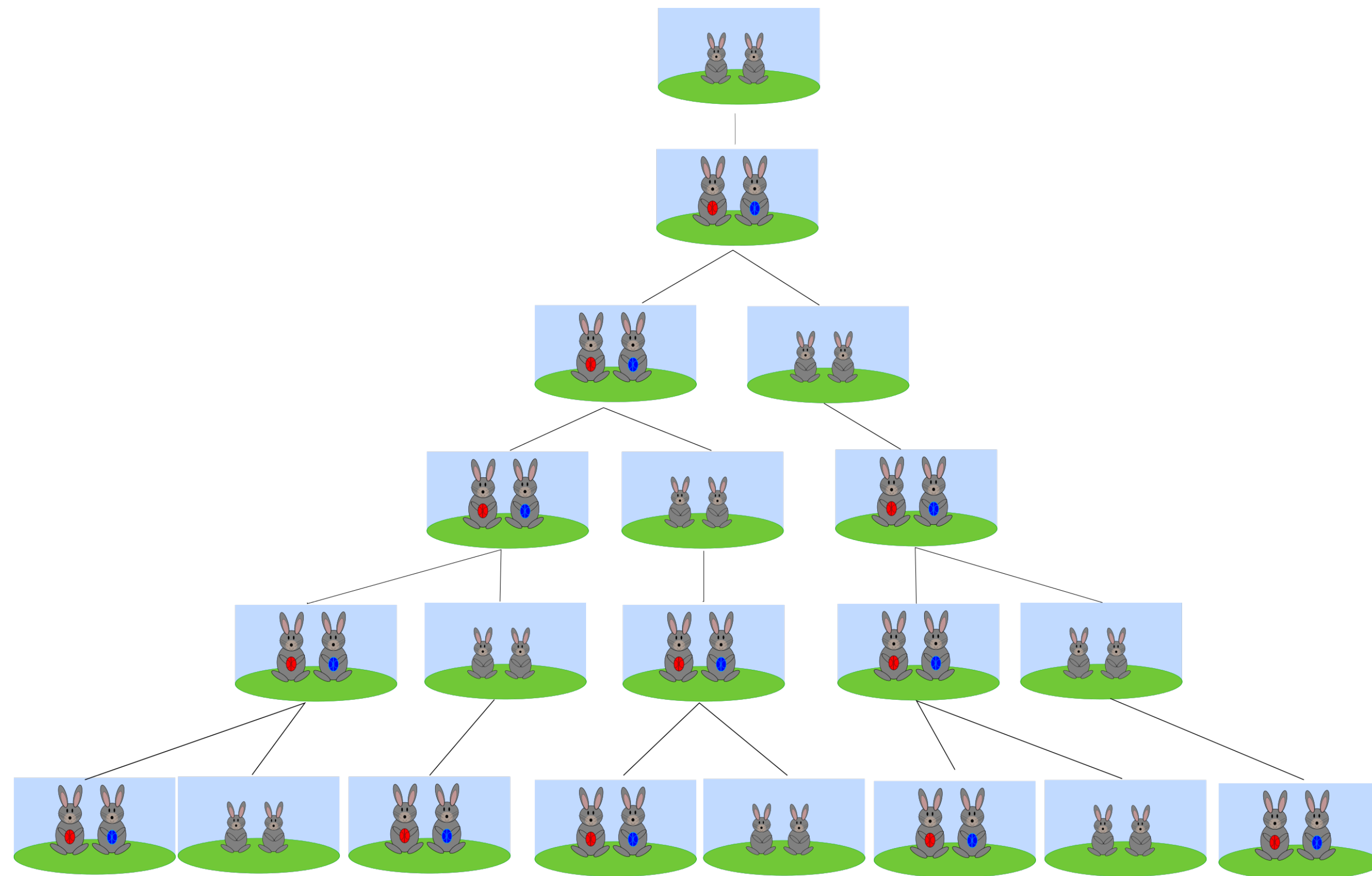
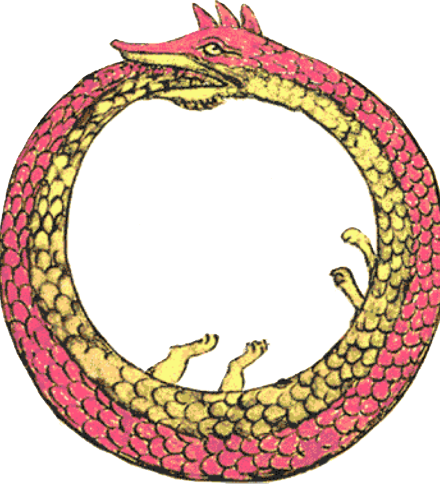
Au fil des générations ...



Au fil des générations ...



Au fil des générations ...



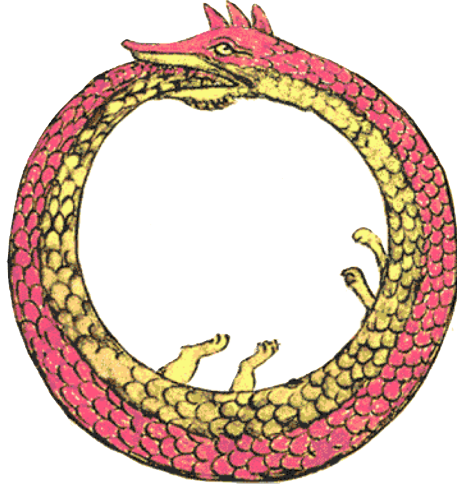
$$F(0) = 0$$

$$F(1) = 1$$

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

```
fonction F(n) (version récursive)
  si n vaut 0 ou 1 alors
    retourner n
  sinon
    retourner  $F(n-1) + F(n-2)$ 
  fin si
```

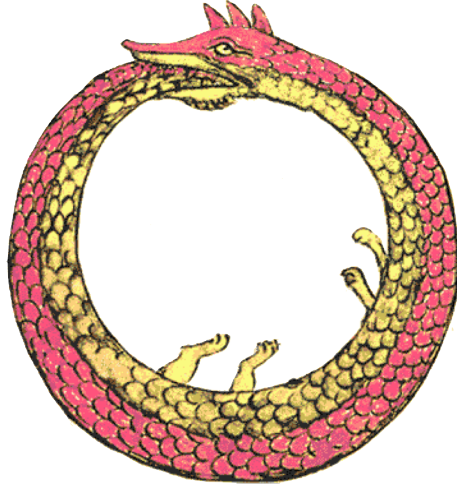
Traçons l'exécution ...



```
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)
```

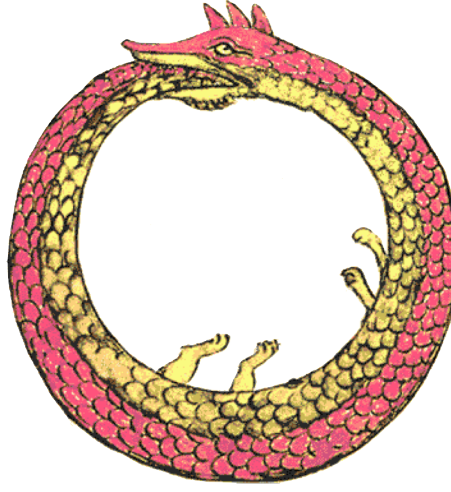
Traçons l'exécution ...



```
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(0)+f(1)
    retourner f(3)+f(2)
```

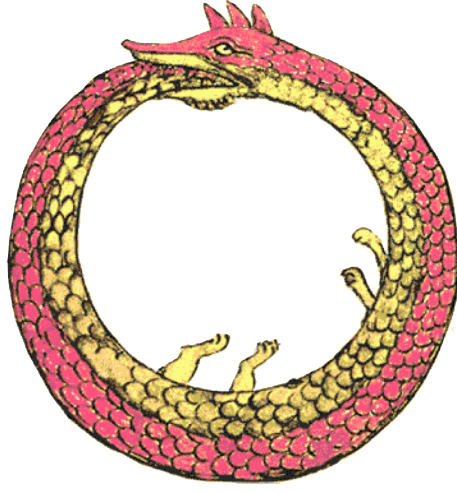
Traçons l'exécution ...



```
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(0)+f(1)
        Calculer f(1)
          retourner 1
        retourner f(2)+f(1)
      Calculer f(2)
        Calculer f(1)
          retourner 1
        Calculer f(0)
          retourner 1
        retourner f(0)+f(1)
      retourner f(3)+f(2)
```

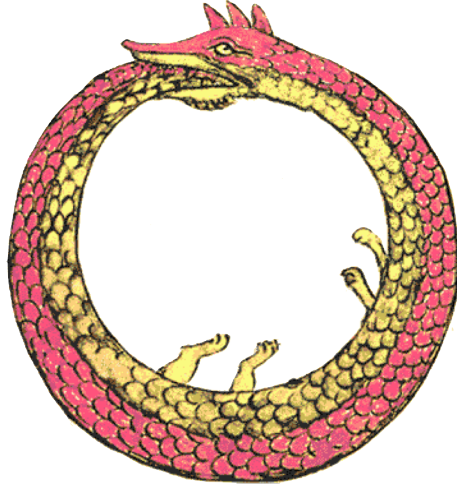
Traçons l'exécution ...



```
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(0)+f(1)
    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(0)+f(1)
  retourner f(3)+f(2)
```

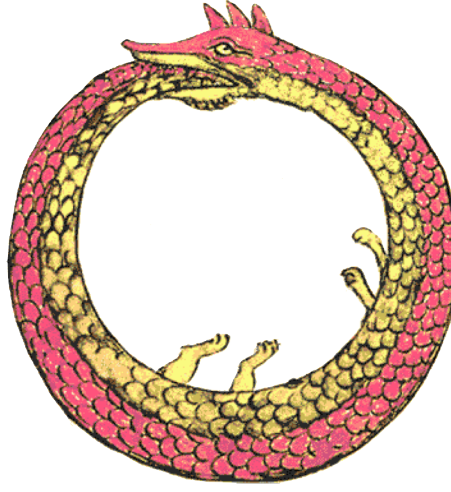
Traçons l'exécution ...



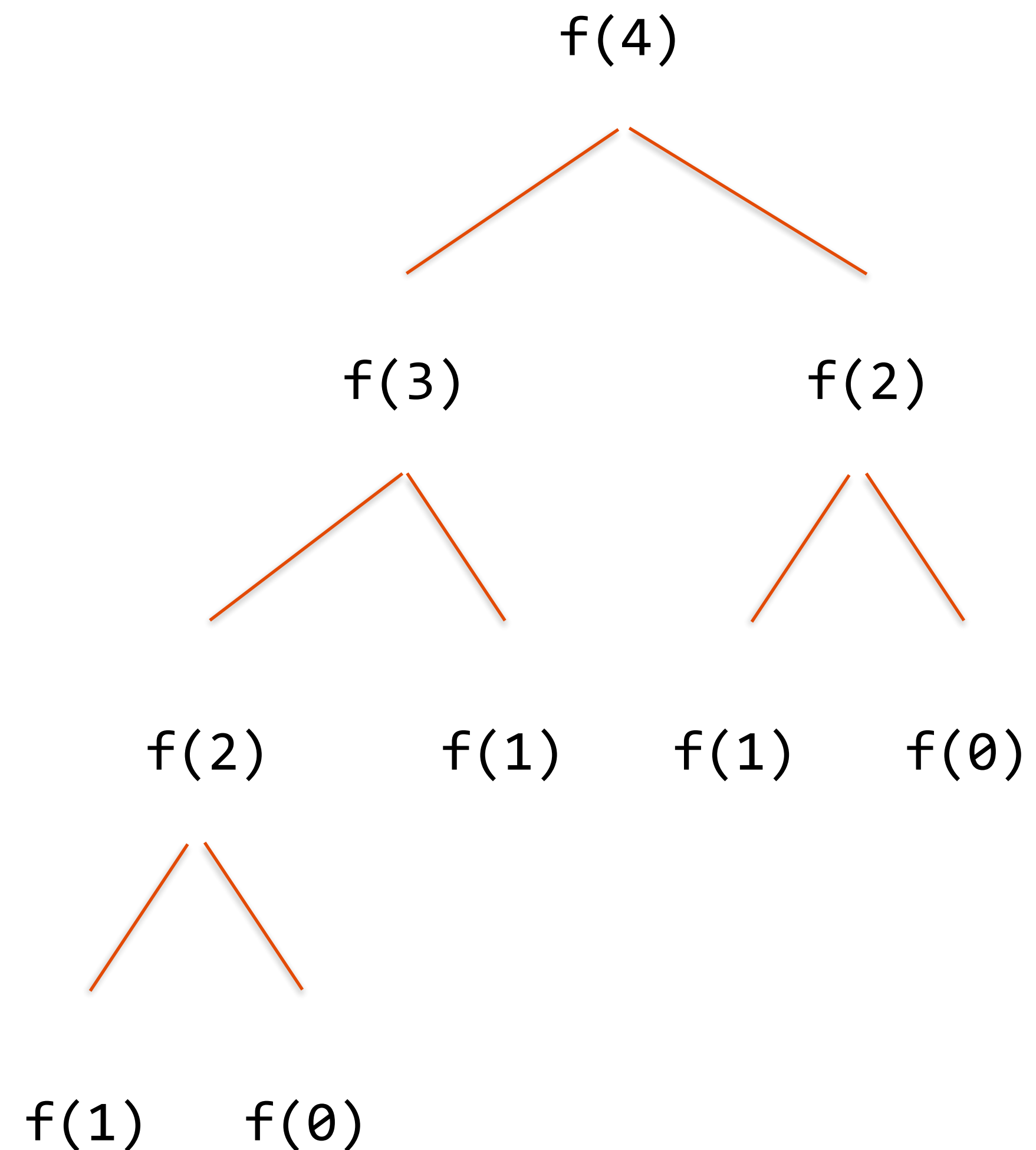
```
fonction F(n) (version récursive)
  si n vaut 0 ou 1 alors
    retourner n
  sinon
    retourner F(n-1) + F(n-2)
  fin si
```

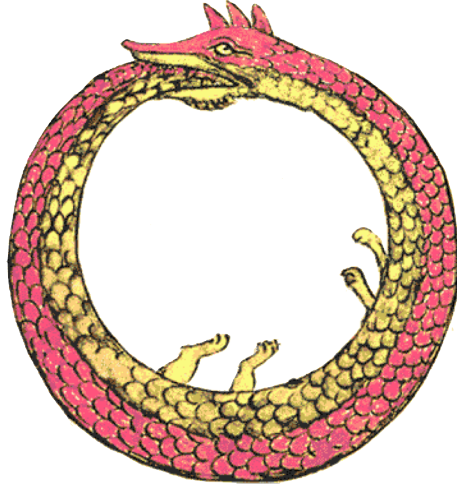
```
Calculer f(4)
  Calculer f(3)
    Calculer f(2)
      Calculer f(1)
        retourner 1
      Calculer f(0)
        retourner 0
      retourner f(0)+f(1) = 1
    Calculer f(1)
      retourner 1
    retourner f(2)+f(1) = 2
  Calculer f(2)
    Calculer f(1)
      retourner 1
    Calculer f(0)
      retourner 0
    retourner f(0)+f(1) = 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
```



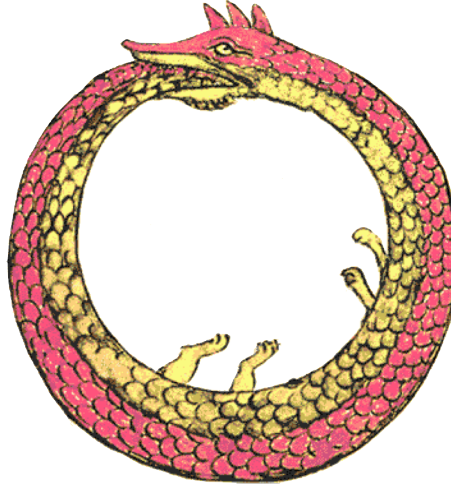


Pour calculer la complexité, nous allons

- Compter le nombre d'appels récuratifs
- Découvrir que ce nombre est proportionnel à $F(n)$
- Trouver une expression non-réursive de $F(n)$

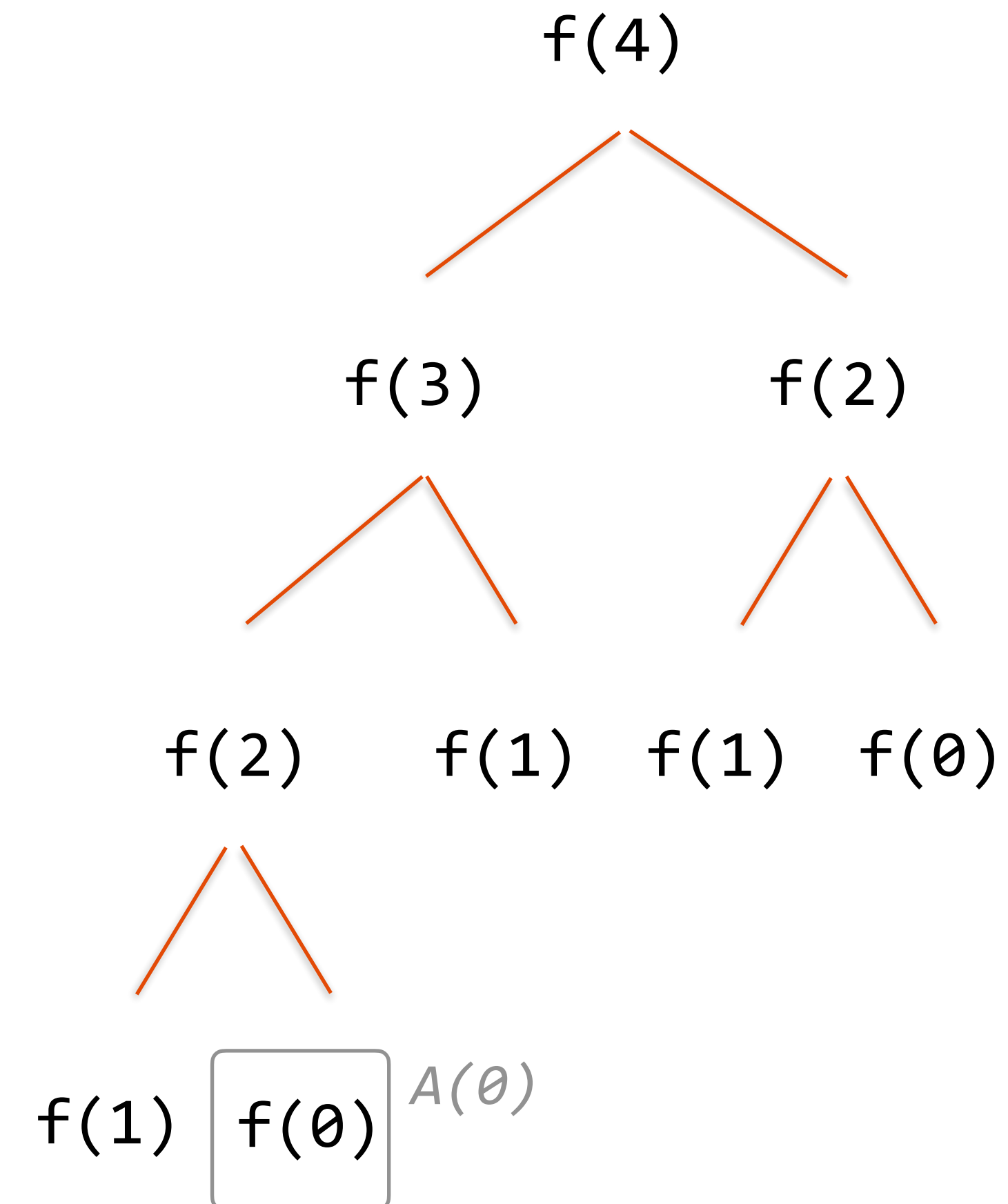
Cette expression sera la complexité.

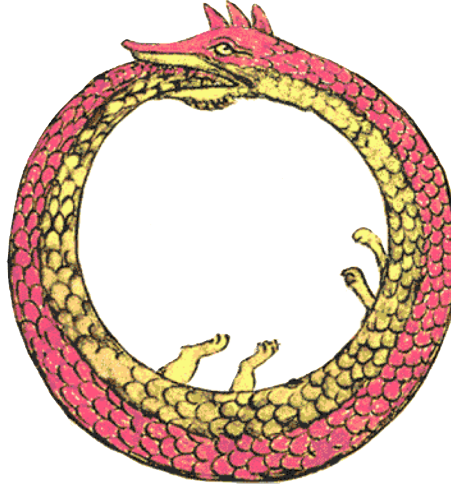
Complexité : Nombre d'appels



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

$$A(0) = 0$$

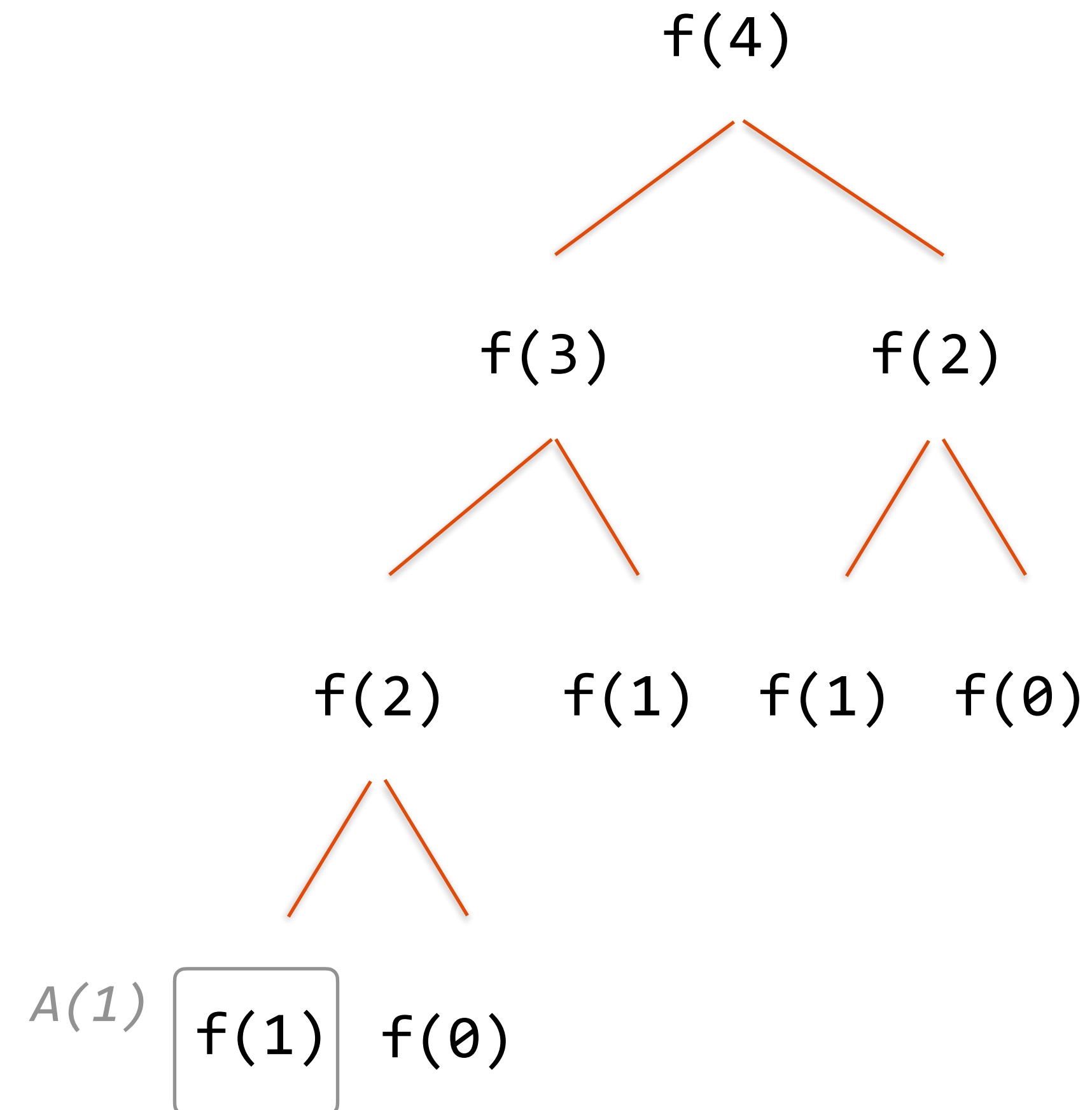


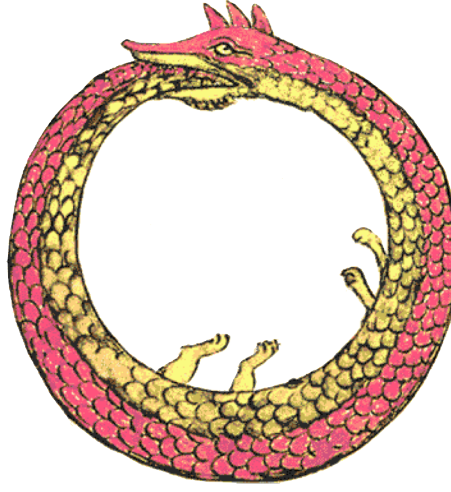


- Quel est le nombre $A(n)$ d'appels récur­sifs effectués pour calculer $F(n)$?

$$A(0) = 0$$

$$A(1) = 0$$



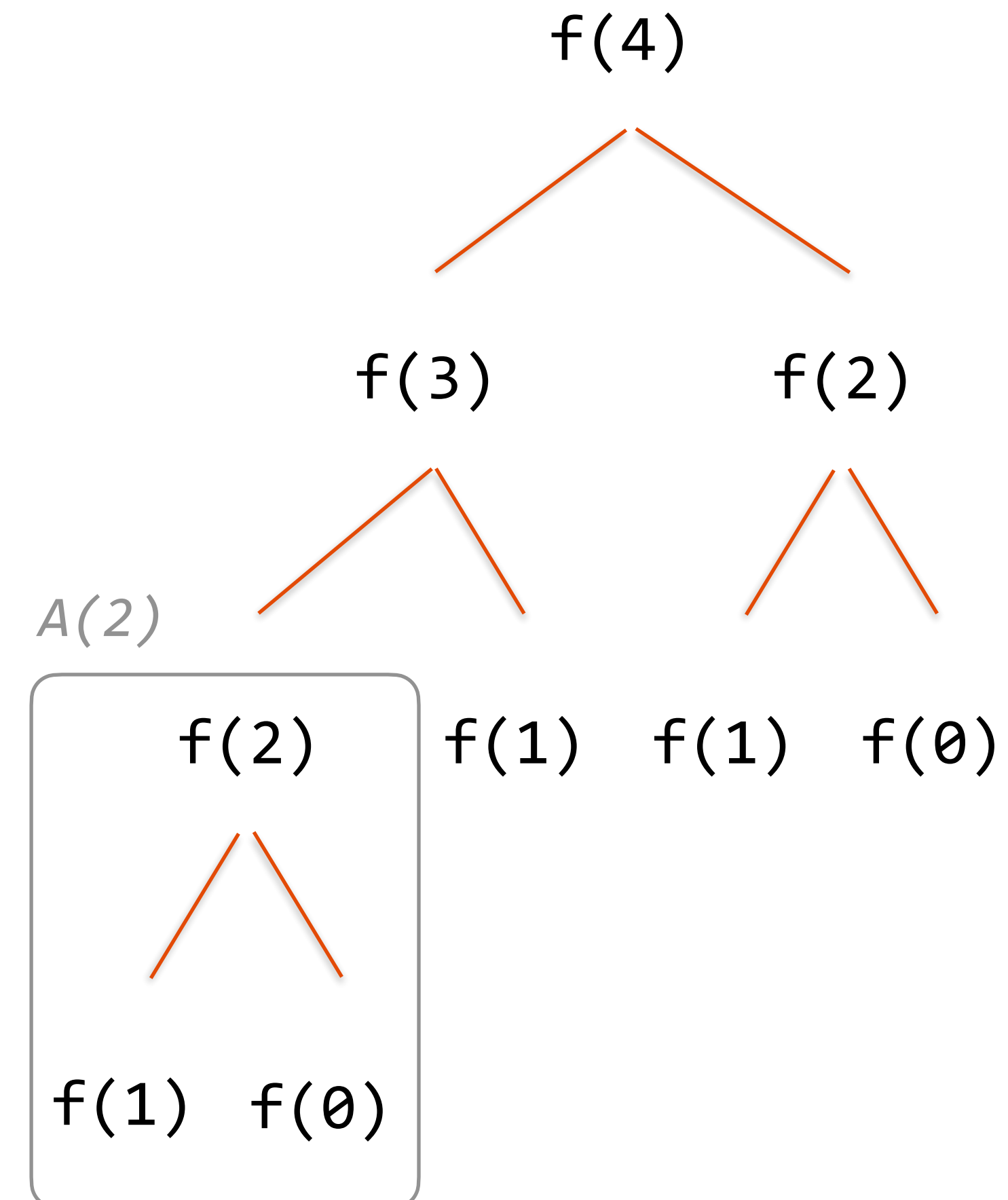


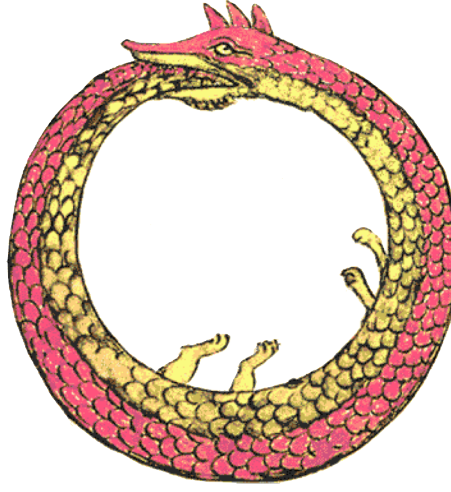
- Quel est le nombre $A(n)$ d'appels récurrents effectués pour calculer $F(n)$?

$$A(0) = 0$$

$$A(1) = 0$$

$$A(2) = 2$$





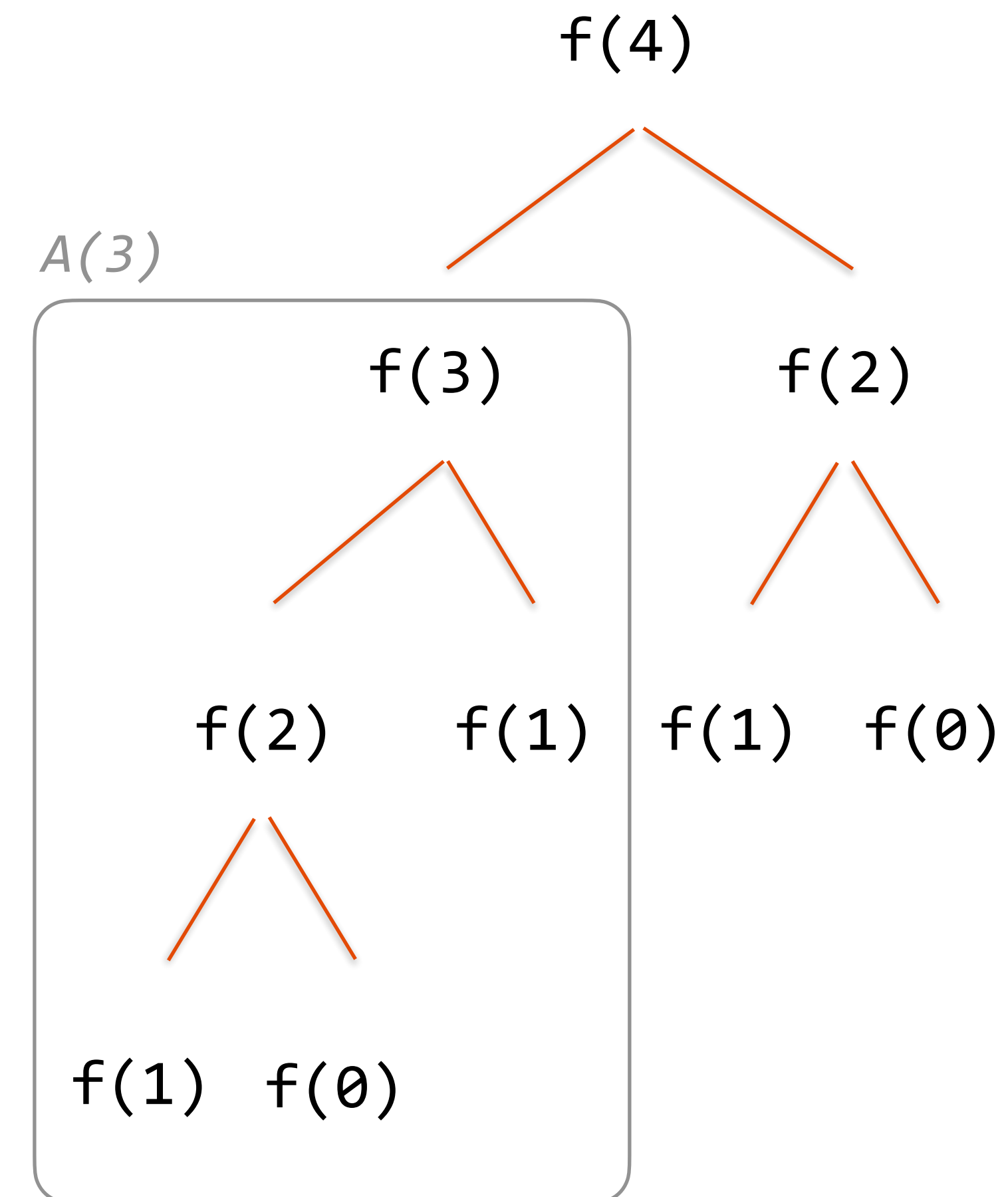
- Quel est le nombre $A(n)$ d'appels récur­sifs effectués pour calculer $F(n)$?

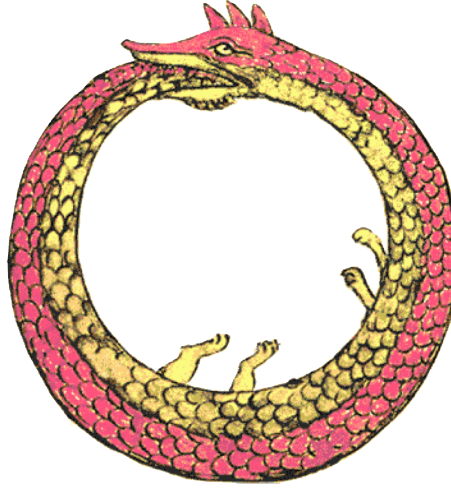
$$A(0) = 0$$

$$A(1) = 0$$

$$A(2) = 2$$

$$A(3) = 2 + A(2) + A(1) = 4$$





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

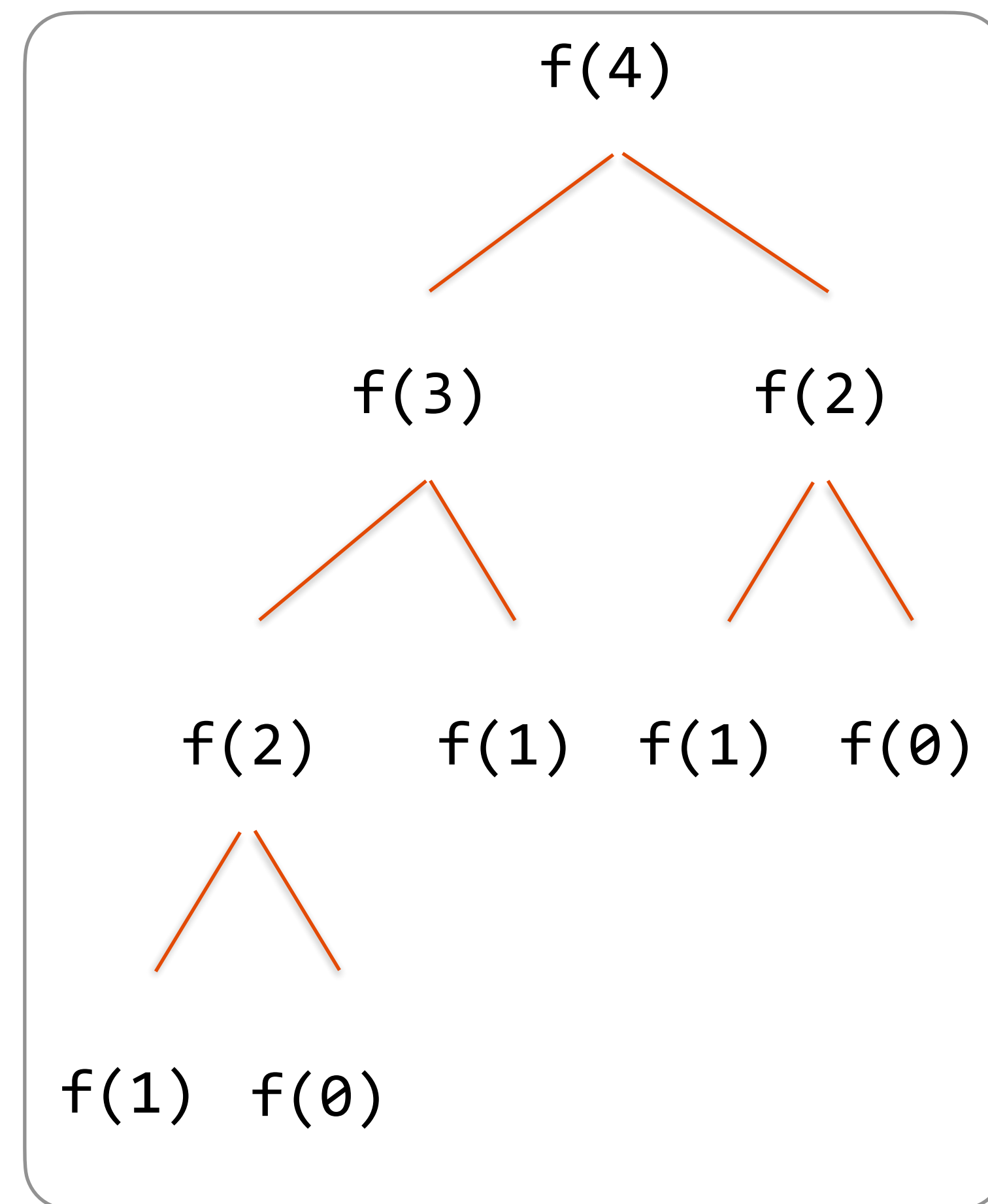
$$A(0) = 0$$

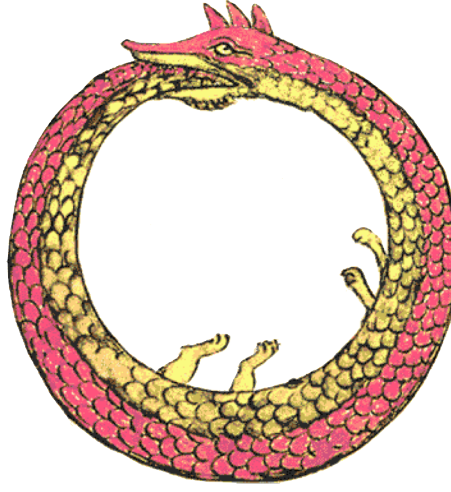
$$A(1) = 0$$

$$A(2) = 2$$

$$A(3) = 2 + A(2) + A(1) = 4$$

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

 $A(4)$ 



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

$$A(0) = 0$$

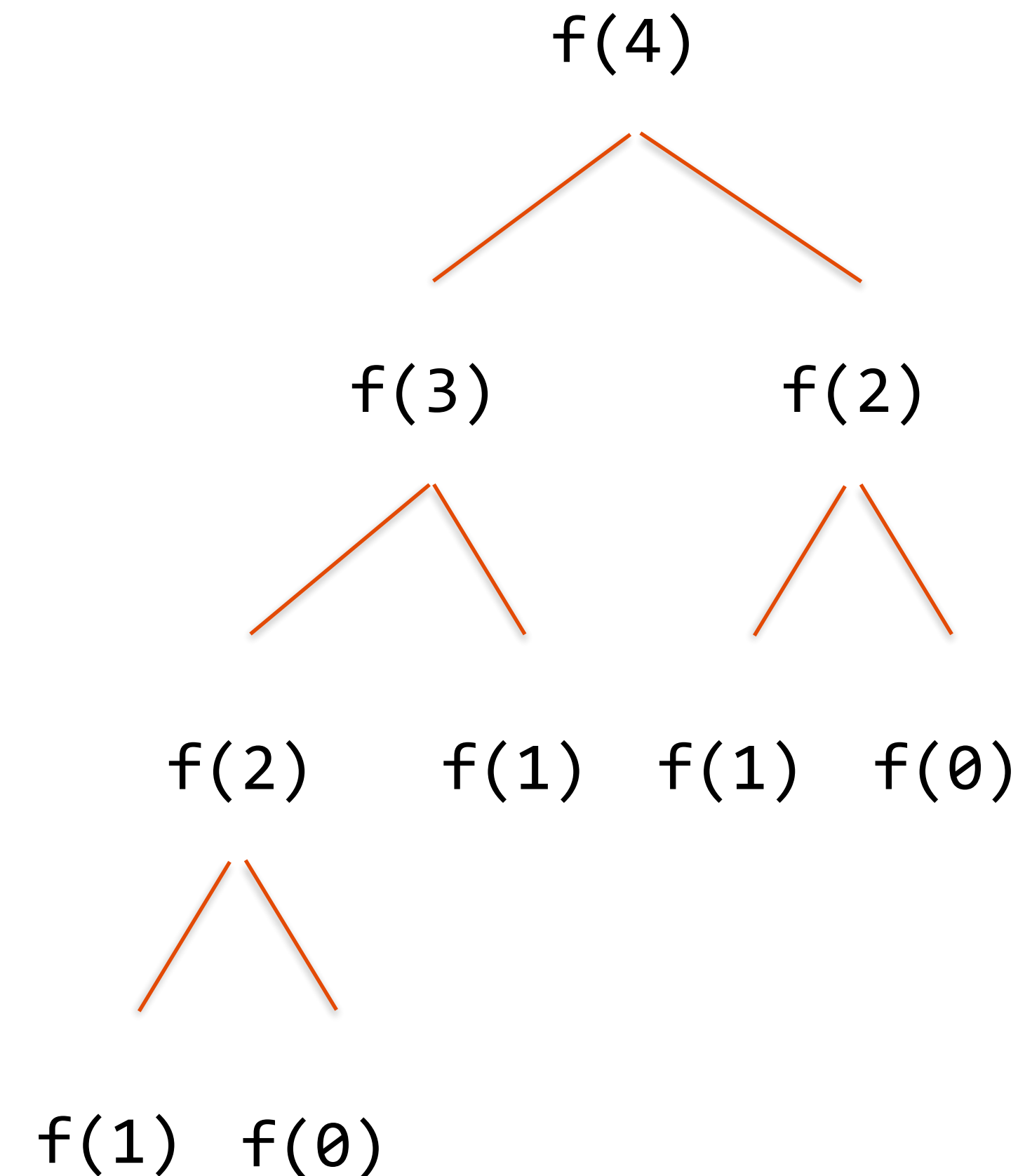
$$A(1) = 0$$

$$A(2) = 2$$

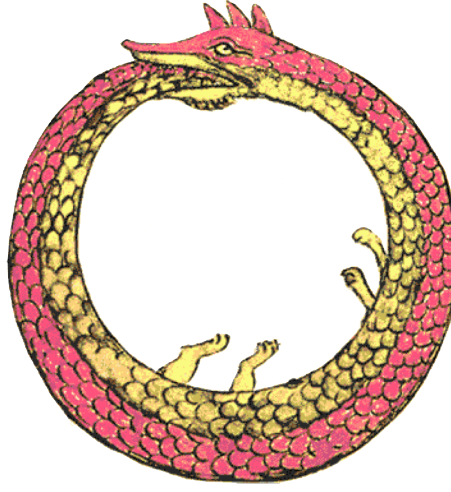
$$A(3) = 2 + A(2) + A(1) = 4$$

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

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



Complexité : $A(n) = O(F(n))$



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

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

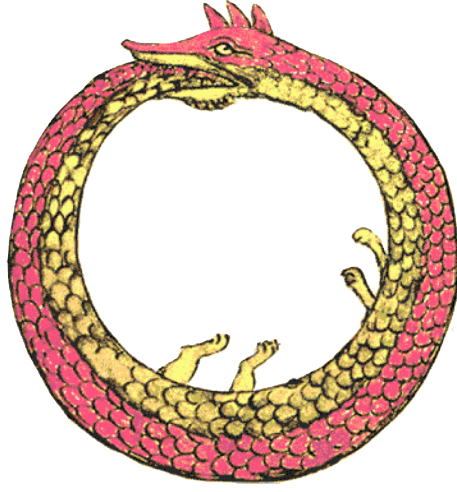
n	0	1	2	3	4	5	6	7	8	9
F(n)	0	1	1	2	3	5	8	13	21	34
2F(n)	0	2	2	4	6	10	16	26	42	68
A(n)	0	0	2	4	8	14	24	40	66	108

Hypothèse : $A(n) = 2 F(n+1) - 2$, pour tout $n > 0$.

Base : Pour $n = 1$, évident. **Récurrence** : Pour tout $m < n$, supposons $A(m-1) = 2F(m) - 2$.

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

$\Rightarrow A$ croît comme F . $\Rightarrow$ L'algorithme a une complexité de $O(F(n+1)) = O(F(n))$.



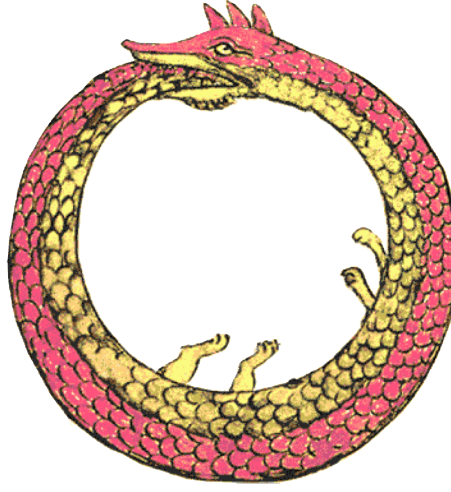
Nous savons maintenant que

- $A(n) = O(F(n))$

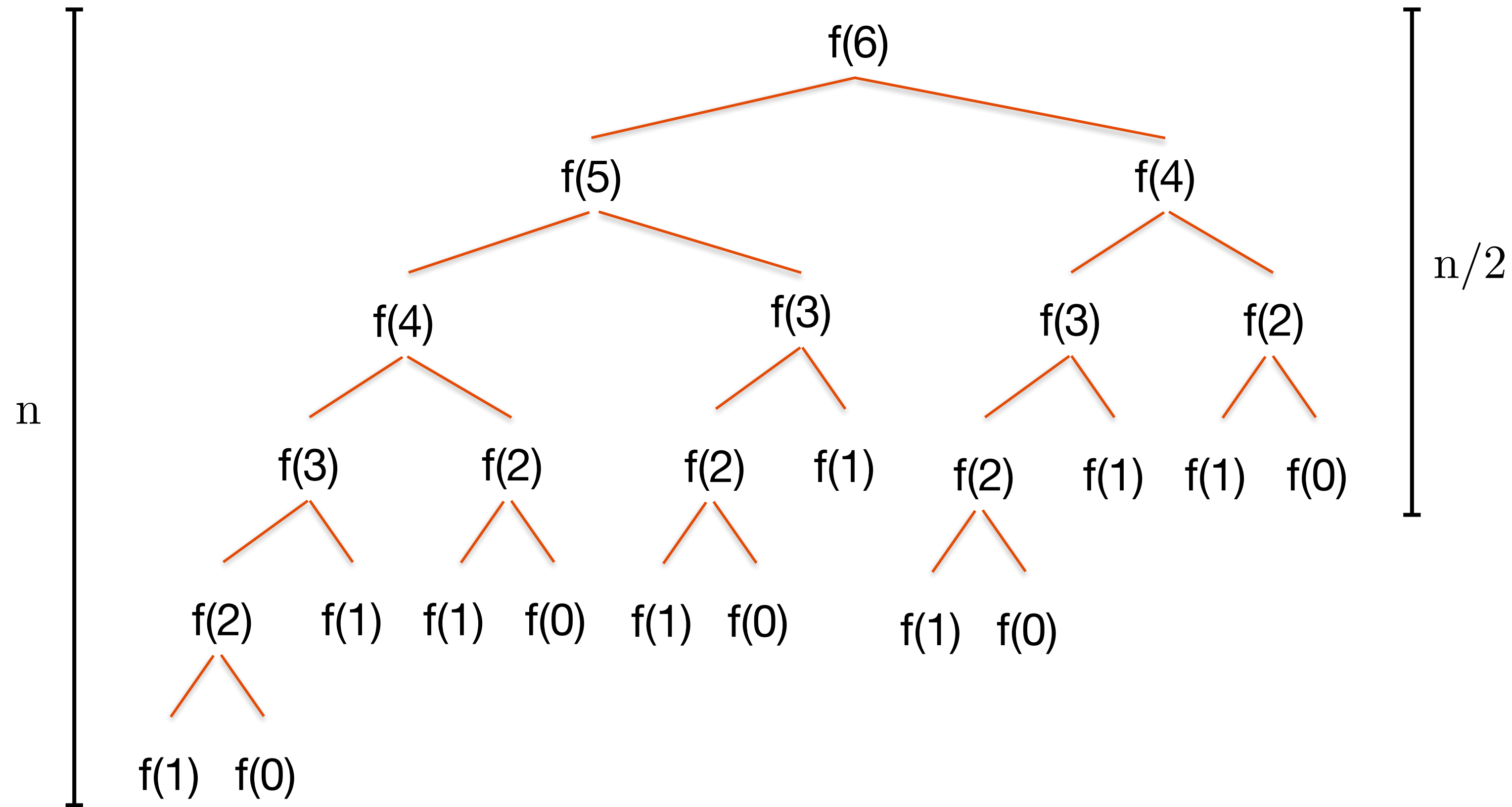
Il nous manque une forme non-réursive de $F(n)$.

Pour ça, nous allons

- Voir que $A(n) = O(x^n)$, pour un certain x ,
- Trouver la valeur de x .



Complexité : $A(n)$ est exponentiel

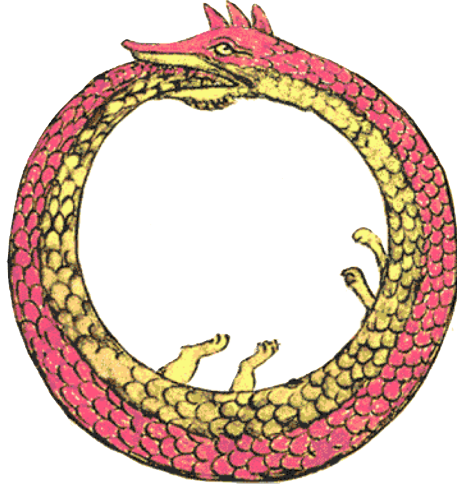


Un arbre de hauteur n contient $\sim 2^n$ noeuds.

$\Rightarrow$ Nombre d'appels récurifs entre 2^n et $2^{(n/2)} = (\sqrt{2})^n \approx 1.1414^n$

$\Rightarrow$ Nombre d'appels récurifs exponentiel : x^n (avec $1.14 < x \leq 2$)

Complexité : Croissance de $F(n)$?



- $A(n)$ croît comme une exponentielle.
- $A(n)$ croît comme $F(n)$.
- Donc $F(n)$ croît comme une exponentielle $\approx x^n$. Que vaut x ?
- On sait que $F(n) = F(n-1) + F(n-2)$

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

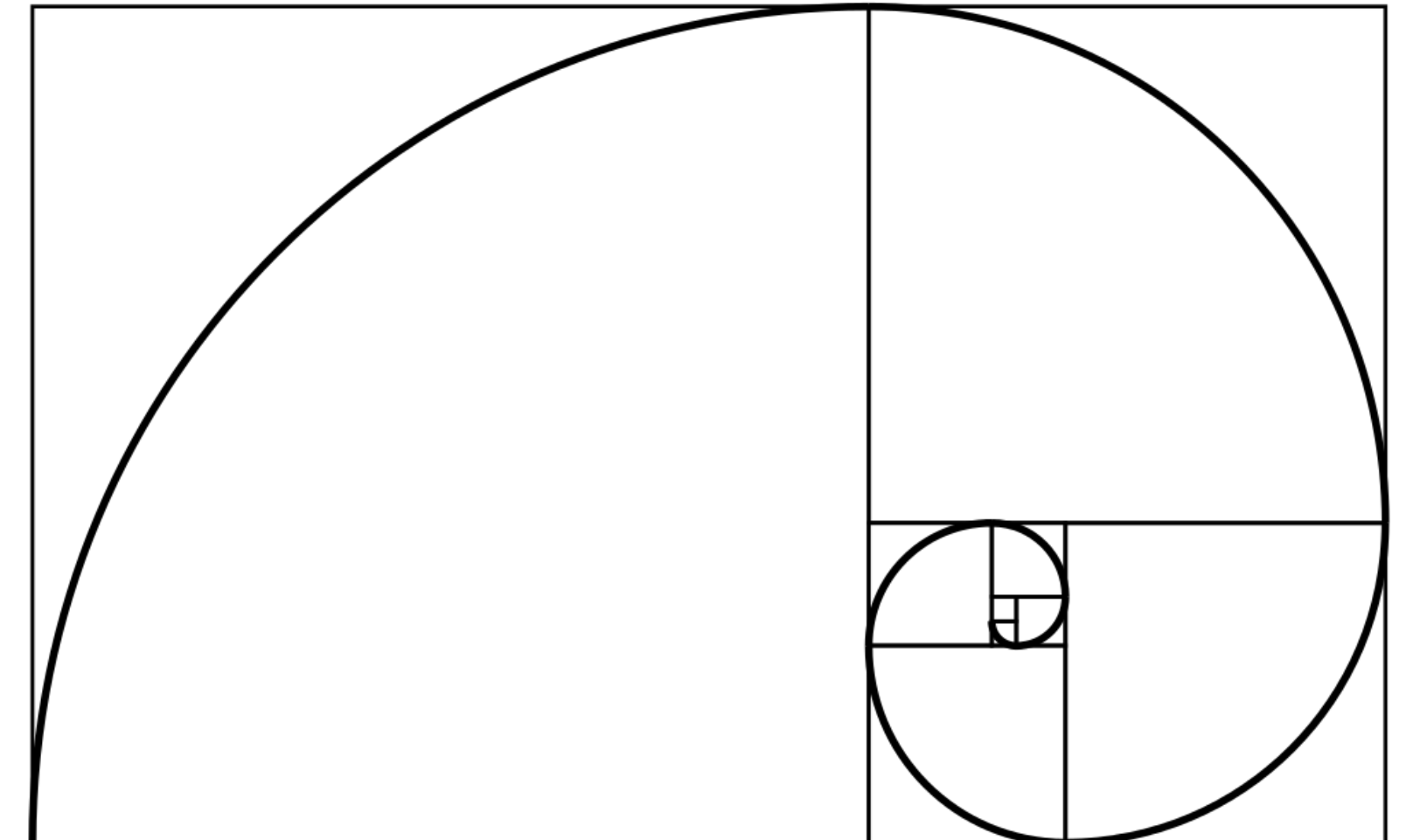
$$x^2 = x + 1$$

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

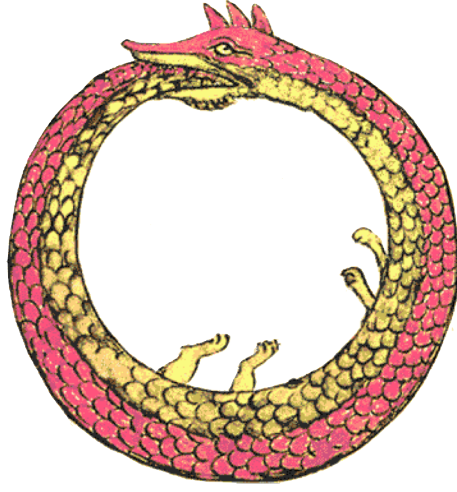
$$x = (1 \pm \sqrt{5}) / 2$$

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

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

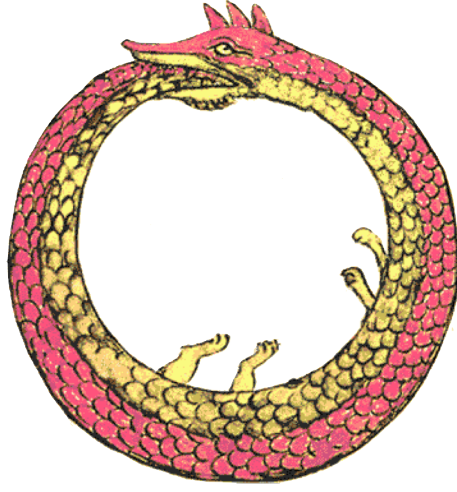


Complexité de l'algorithme



- L'algorithme a une complexité de $A(n)$
- $A(n) = O(F(n))$
- $F(n) = O(\varphi^n)$
- Donc, l'algorithme a une complexité de $O(\varphi^n)$

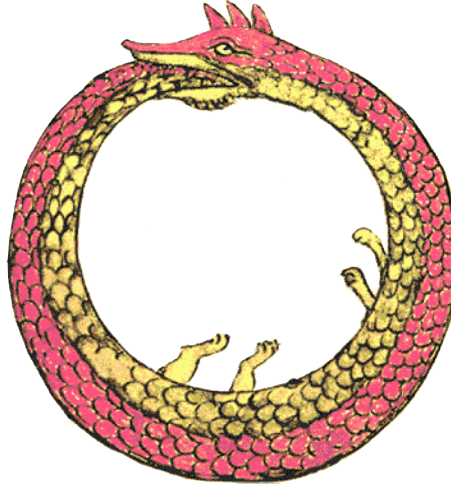
Algorithme d'Euclide



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

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

Algorithme d'Euclide



- De manière générale, pour $a > b$,
 $a \% b = a - nb$, pour un certain $n \geq 1$

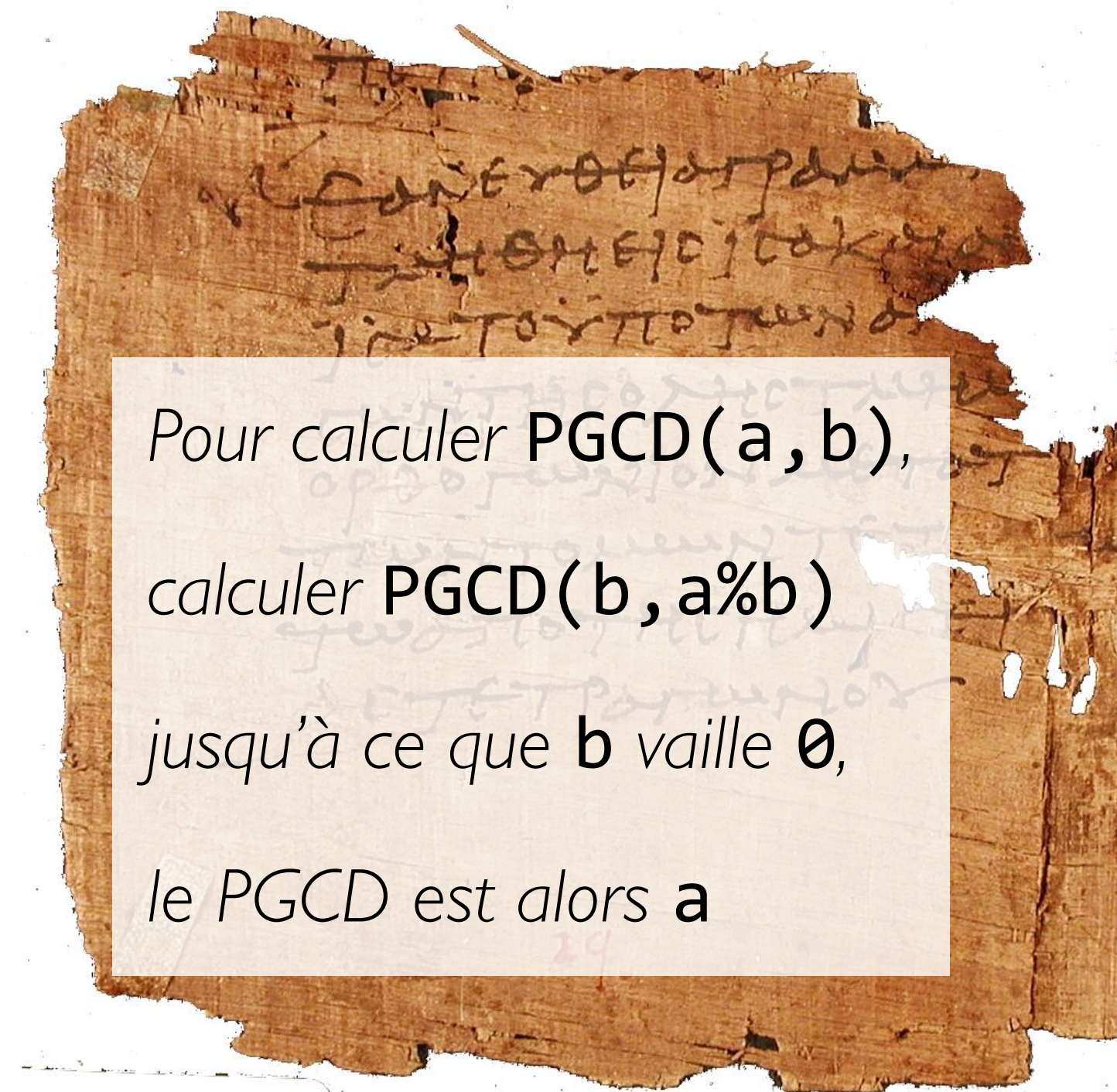
- Dans le pire cas, a et b sont tels que
 $a \% b = a - b$

- La pire suite est donc telle que

$$(a, b) \rightarrow (b, a-b) \rightarrow (a-b, 2b-a) \rightarrow \dots$$

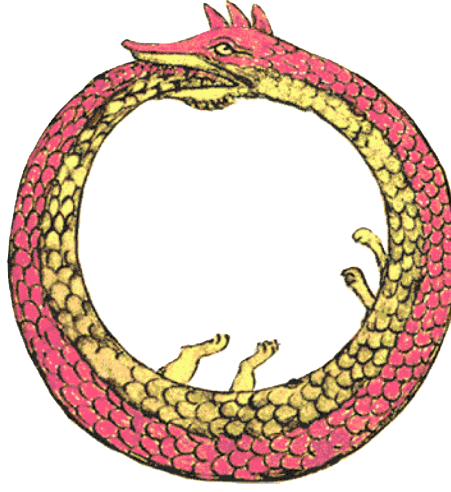
$= (a-b, b - (a-b))$

- La suite de Fibonacci ...

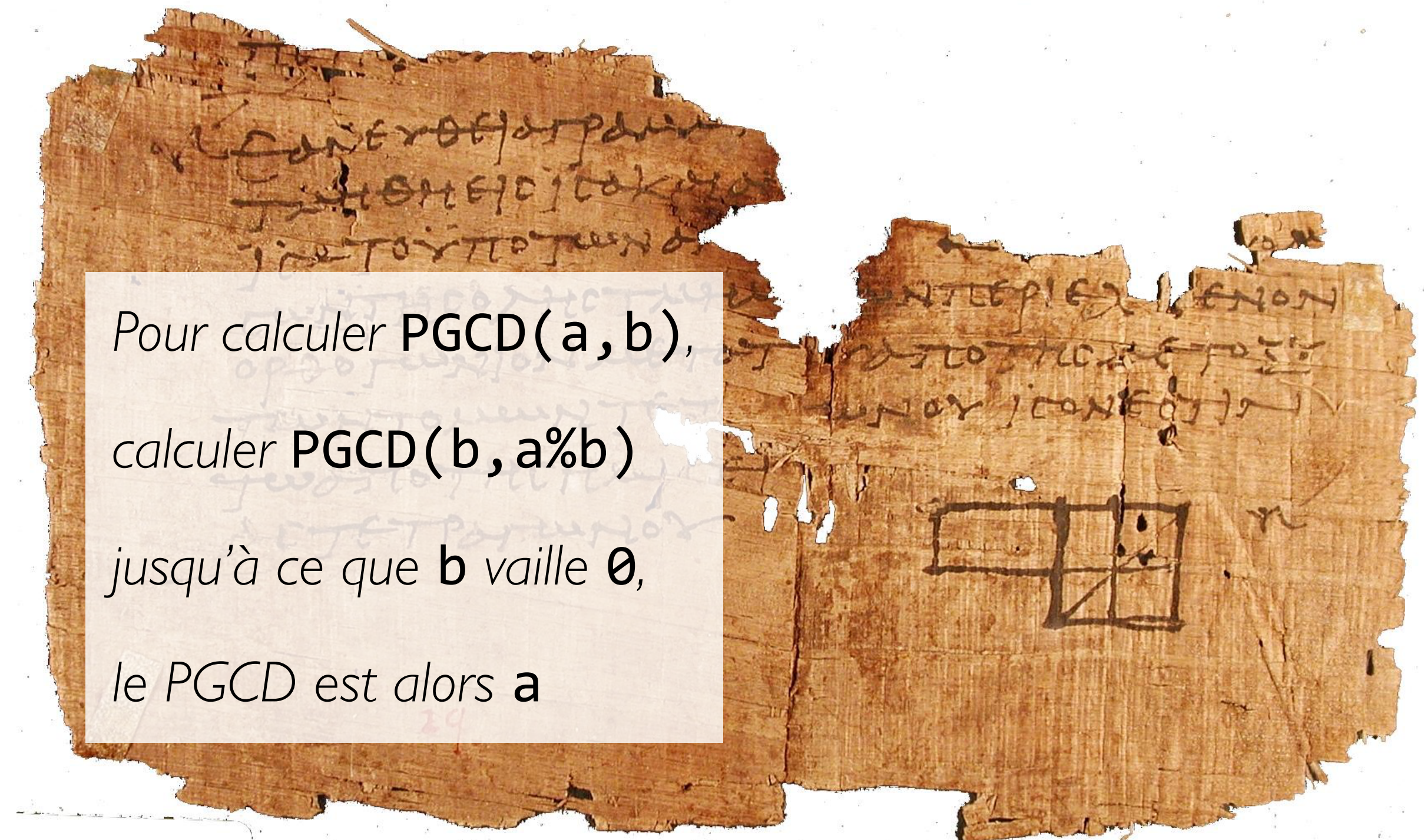


a	b
34	21
21	13
13	8
8	5
5	3
3	2
2	1
1	1
1	0

Algorithme d'Euclide



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



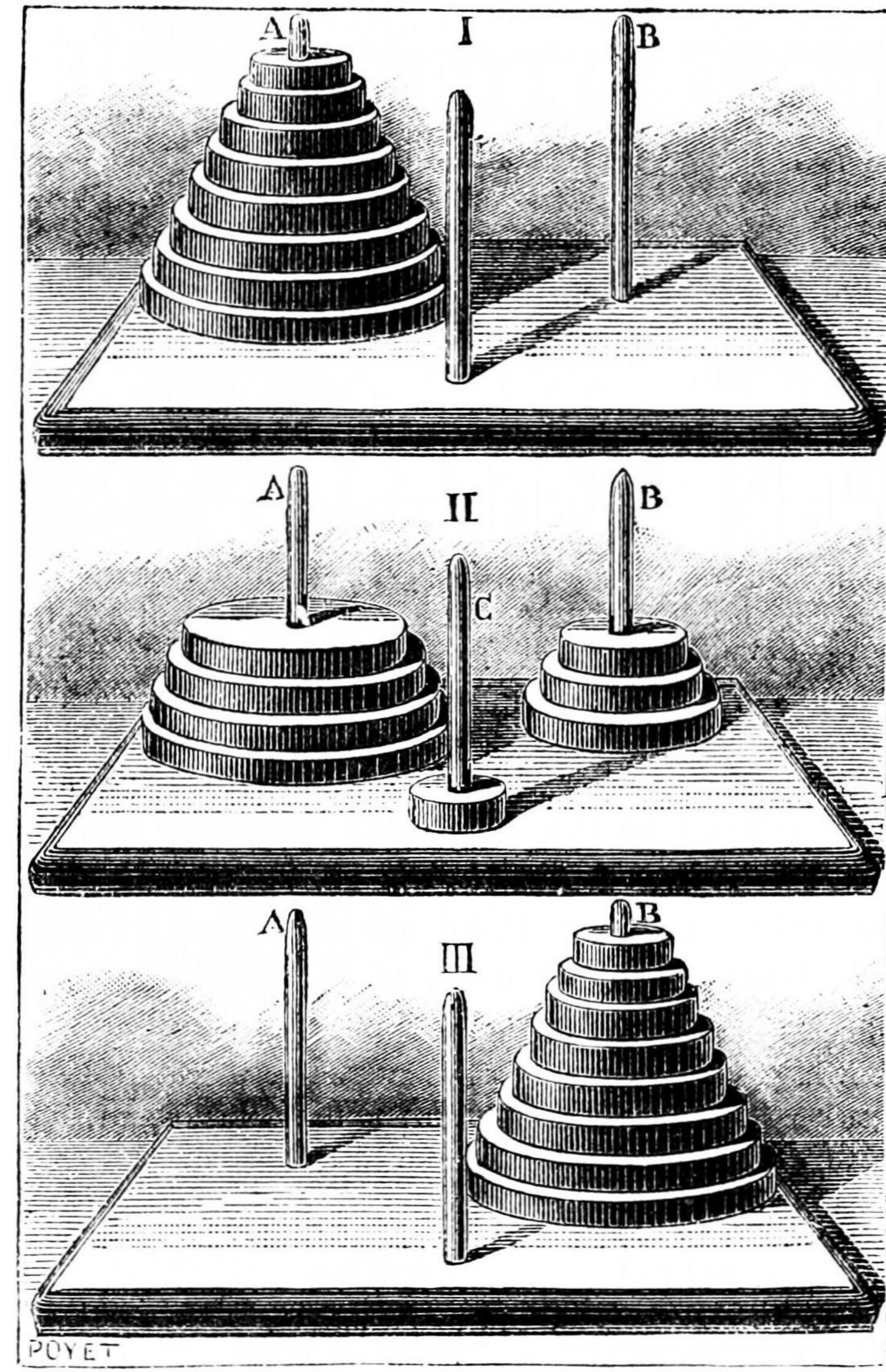
2.3. Tours de Hanoï



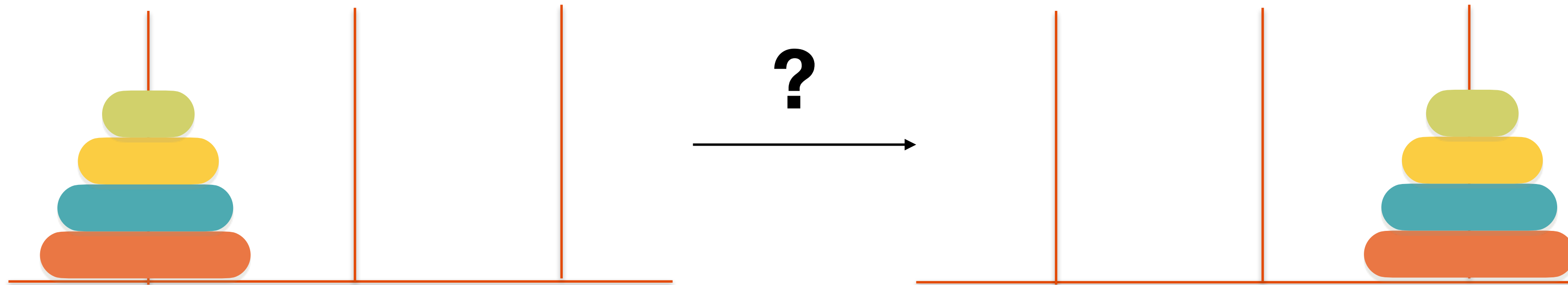
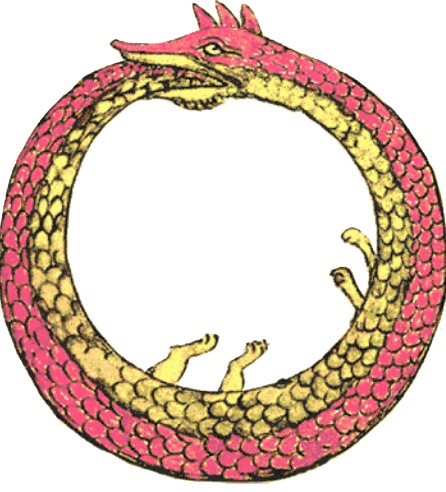
HE^{VD} IG Les tours de Hanoï

« ... 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'écarter 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! »

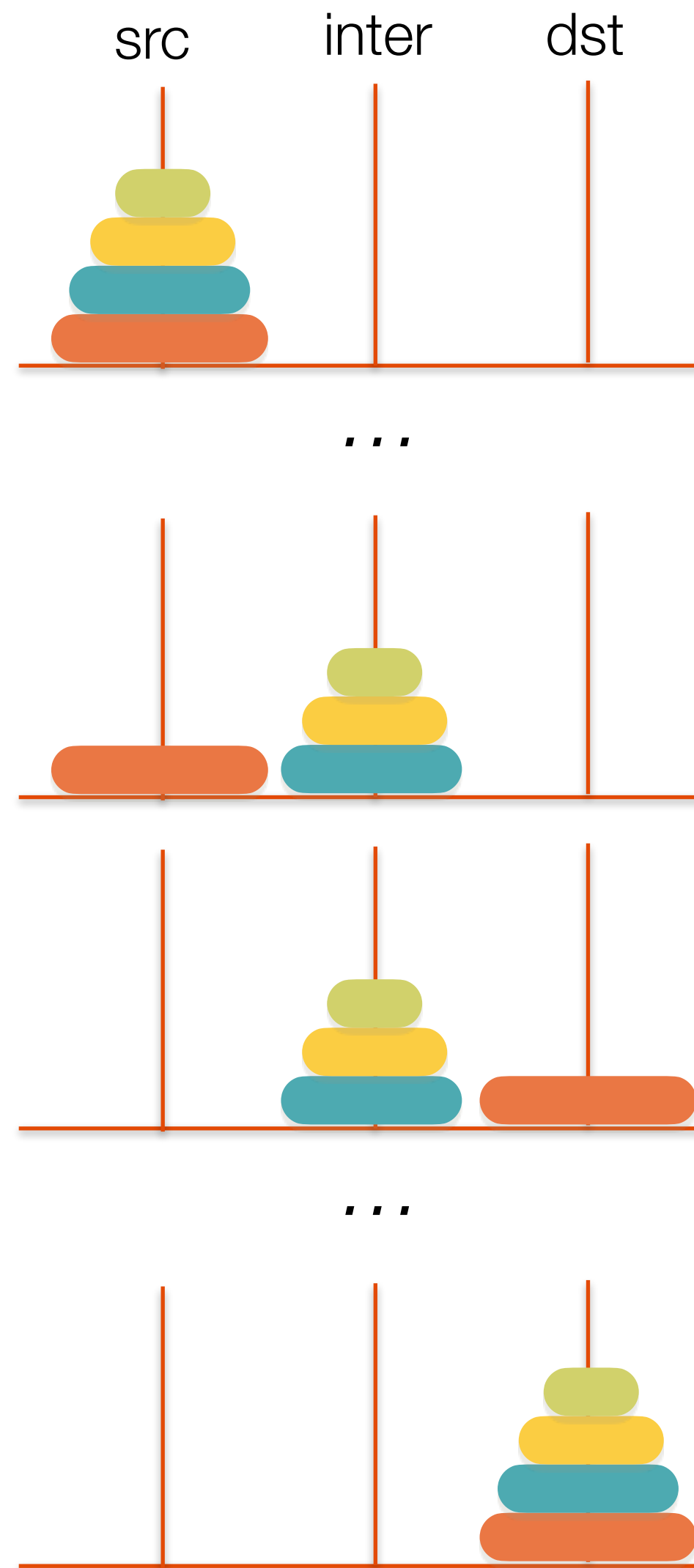
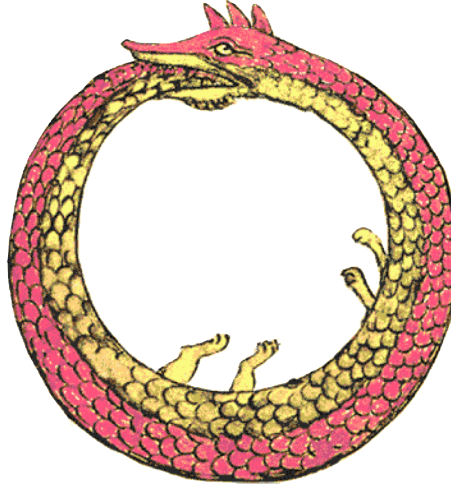


Règles du jeu



- 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



Pour déplacer n disques de gauche (src) à droite (dst) :

Déplacer $n-1$ disques de gauche au milieu

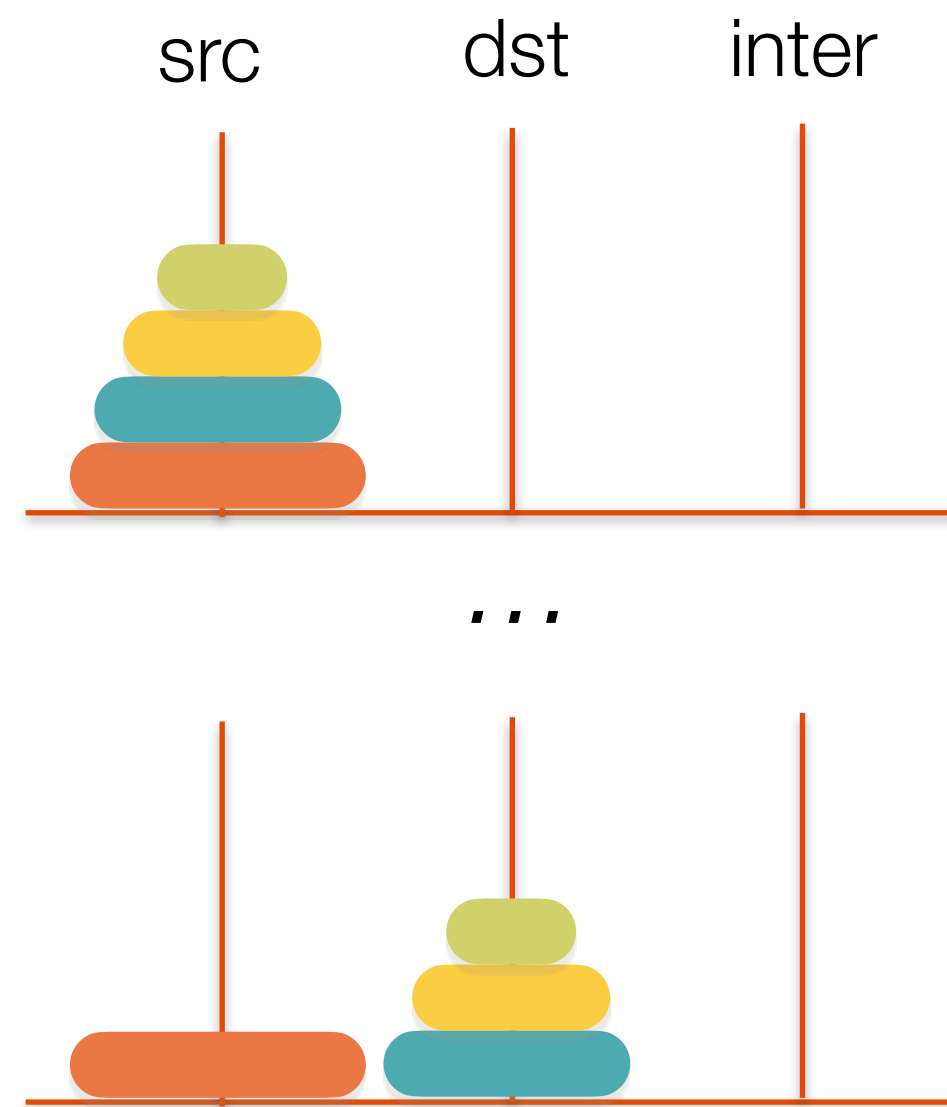
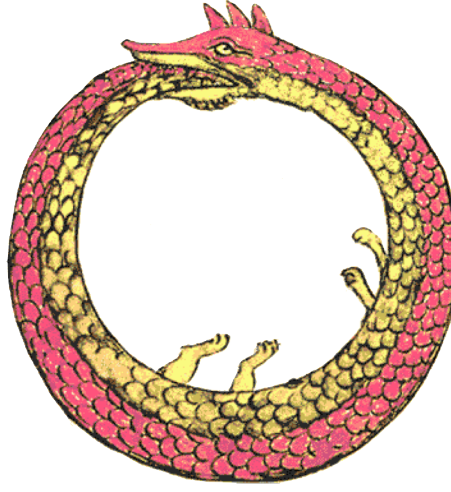
Récuratif

Déplacer le $n^{\text{ième}}$ disque de gauche à droite

Déplacer $n-1$ disques du milieu à droite

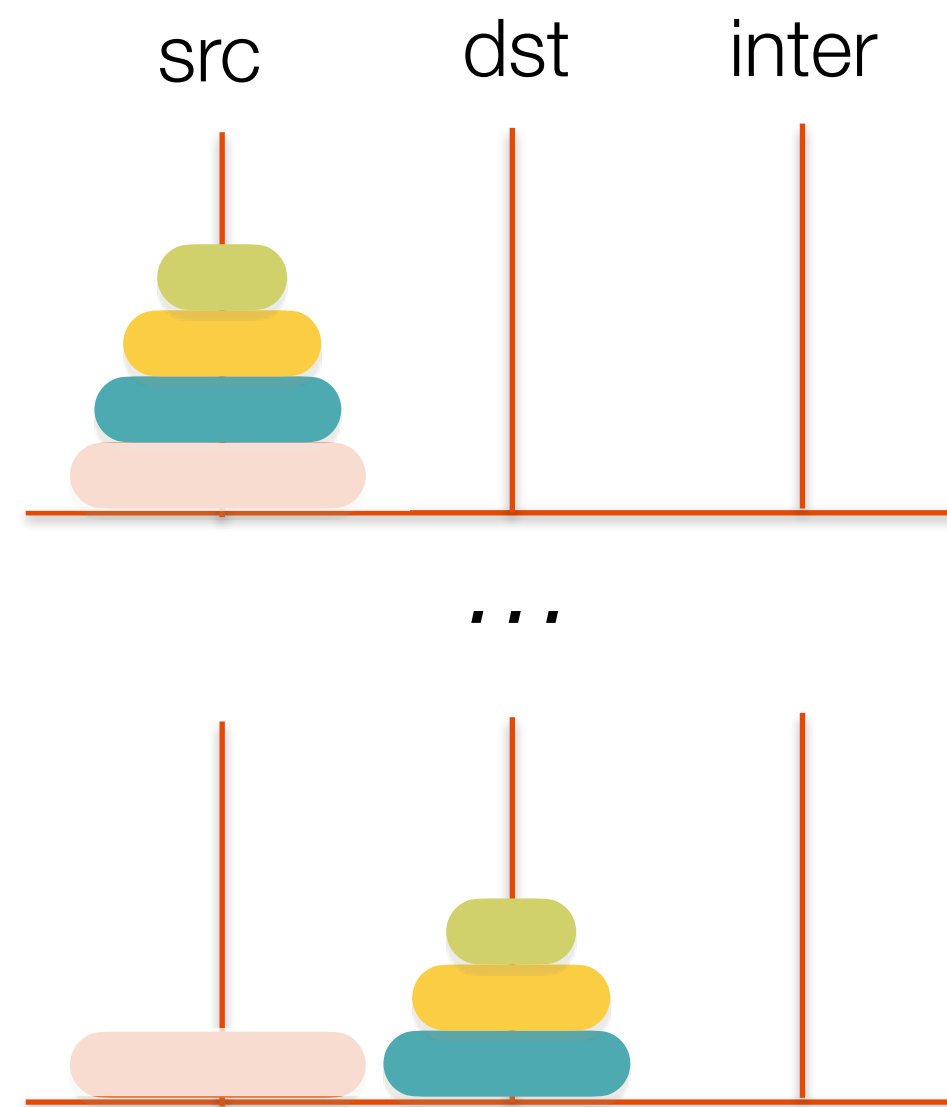
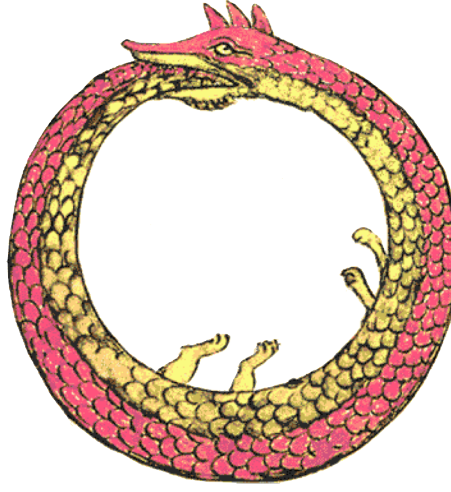
Récuratif

Principe de la solution



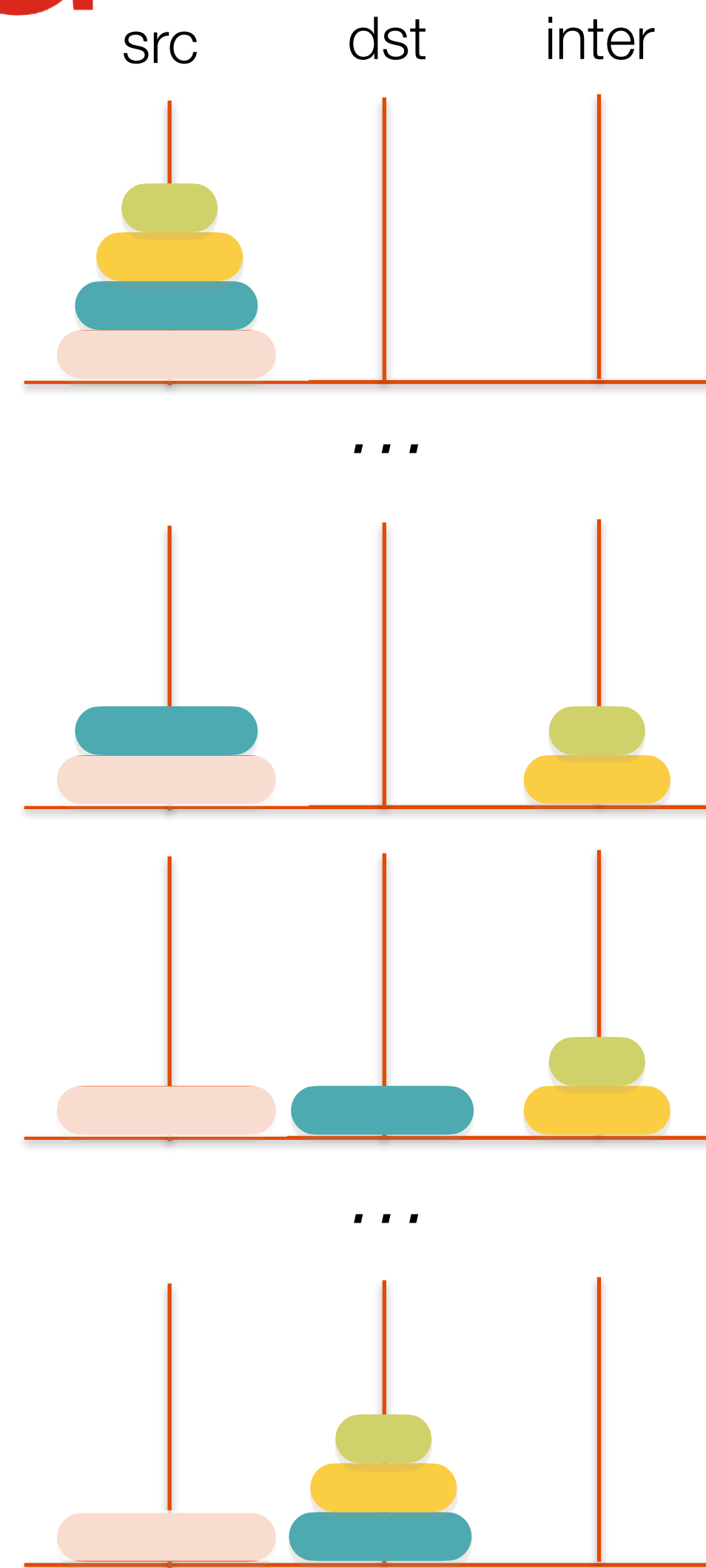
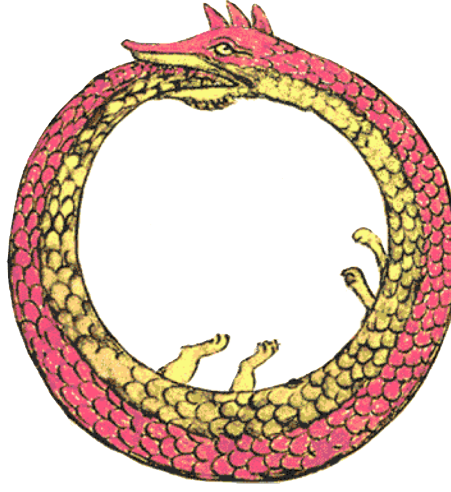
Déplacer n-1 disques de gauche au milieu

Principe de la solution



Déplacer n-1 disques de gauche au milieu

Principe de la solution



Pour déplacer $n-1$ disques de src à dst :

Déplacer $n-2$ disques de src à inter

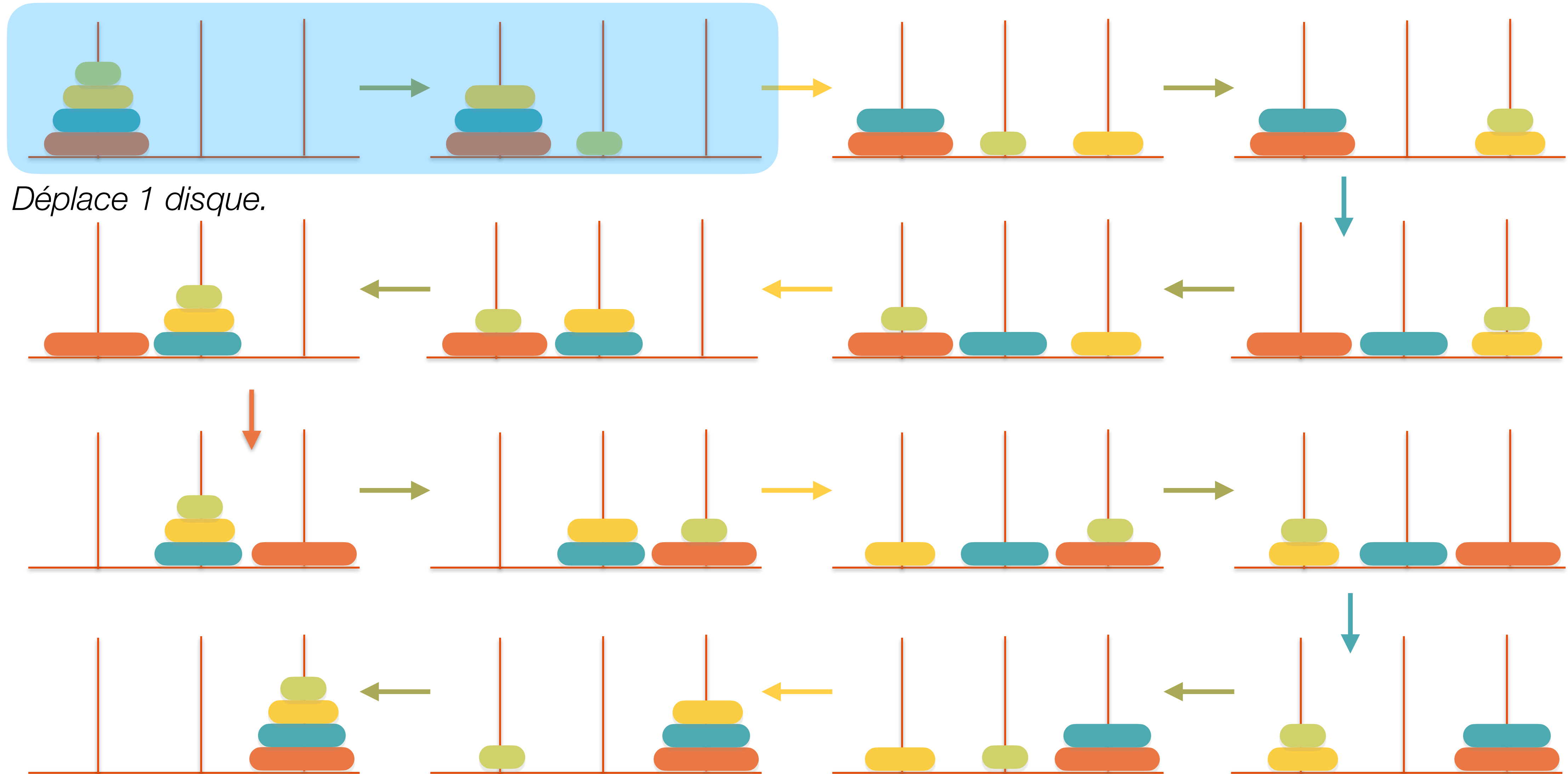
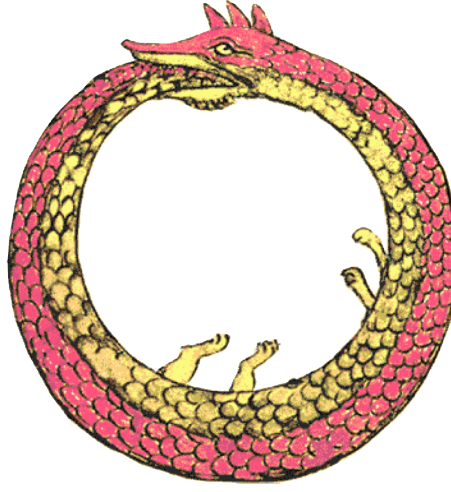
Récuratif

Déplacer le $n-1^{\text{eme}}$ disque de src à dst

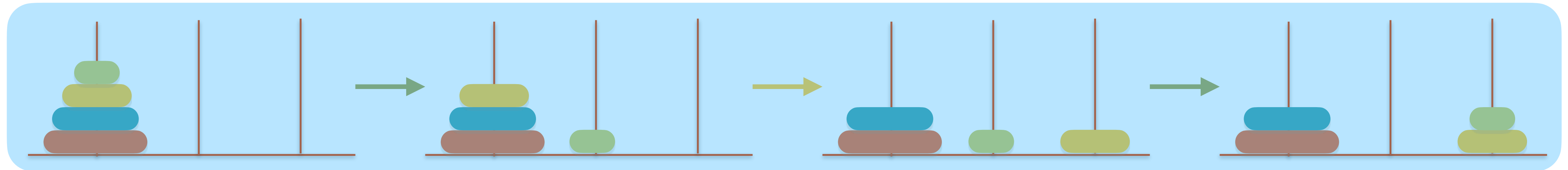
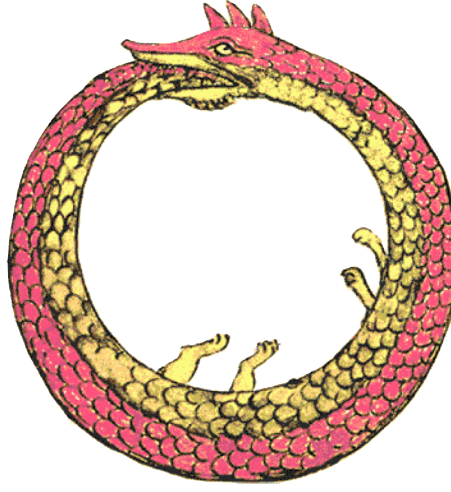
Déplacer $n-2$ disques de inter à dst

Récuratif

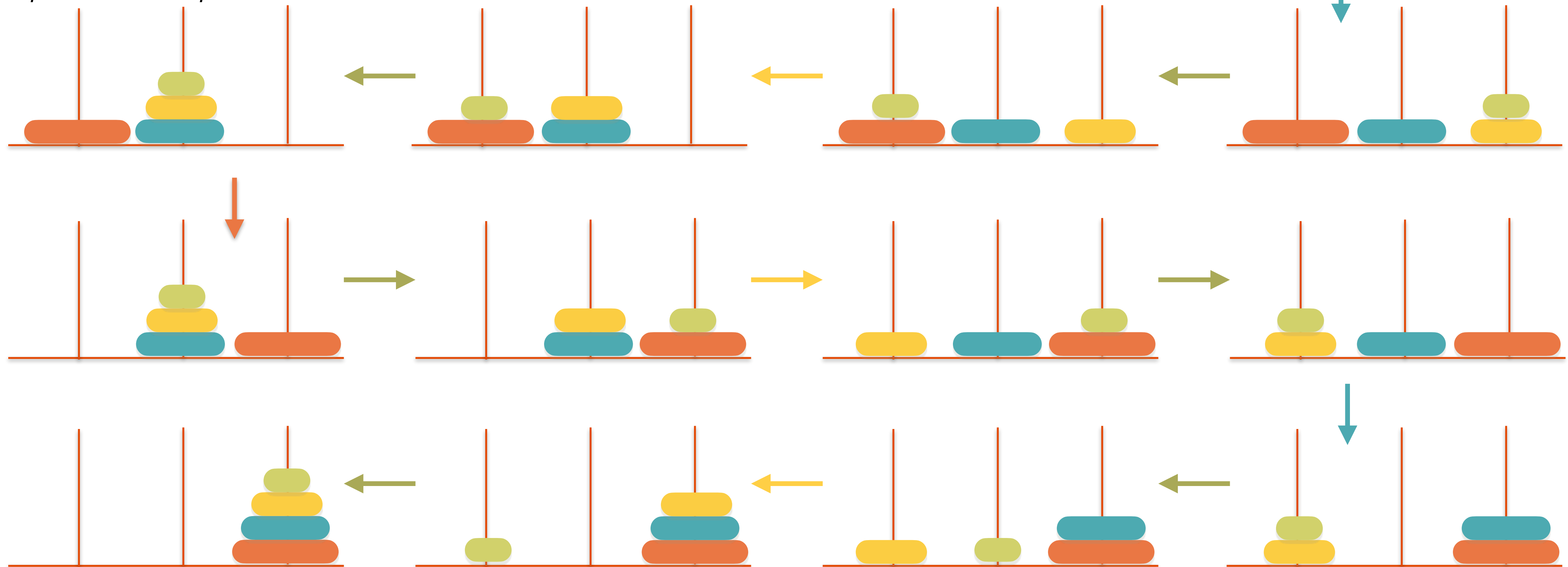
Toutes les étapes pour 4 disques



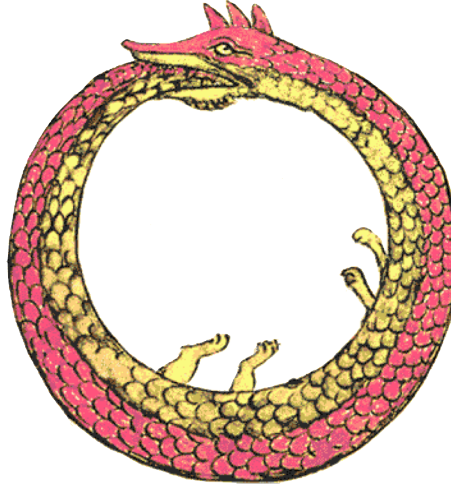
Toutes les étapes pour 4 disques



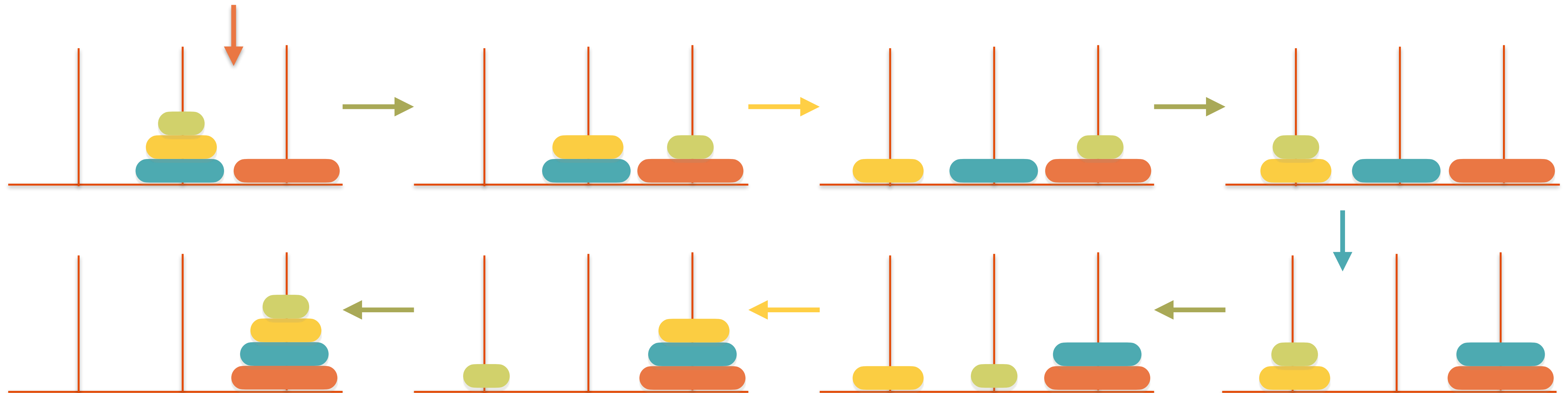
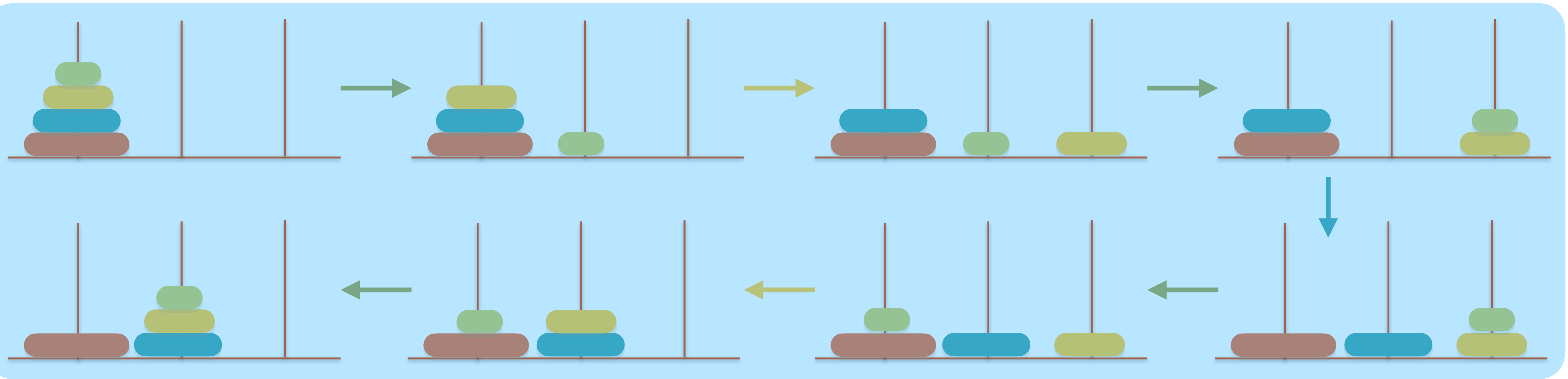
Déplace 2 disques.



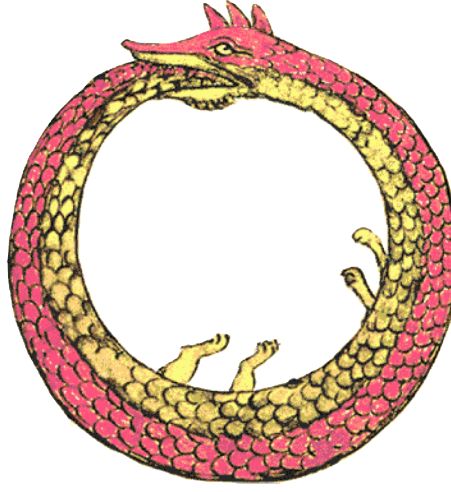
Toutes les étapes pour 4 disques



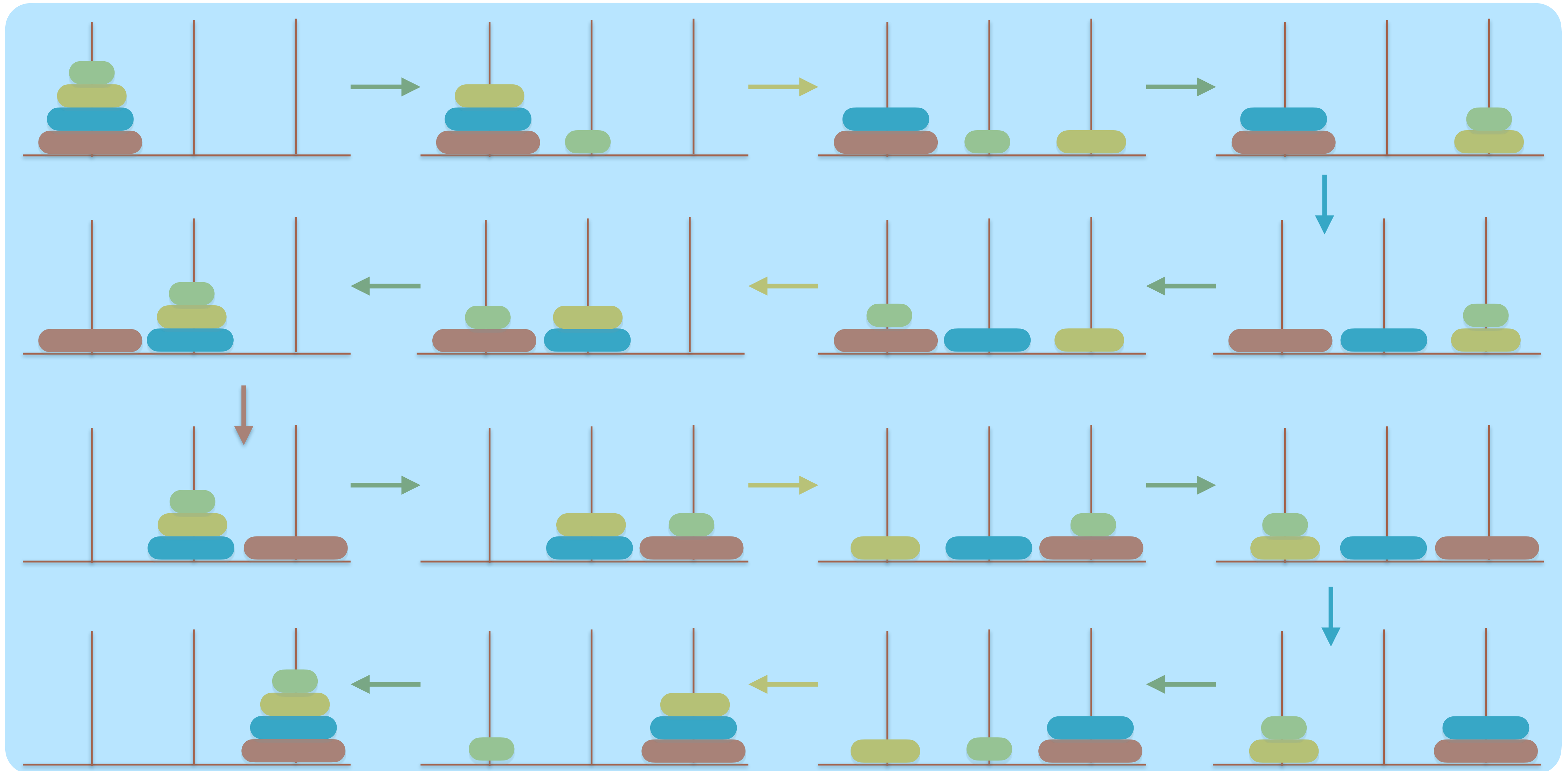
Déplace 3 disques.



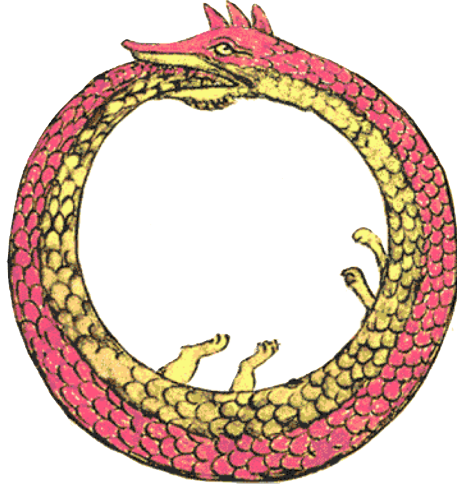
Toutes les étapes pour 4 disques



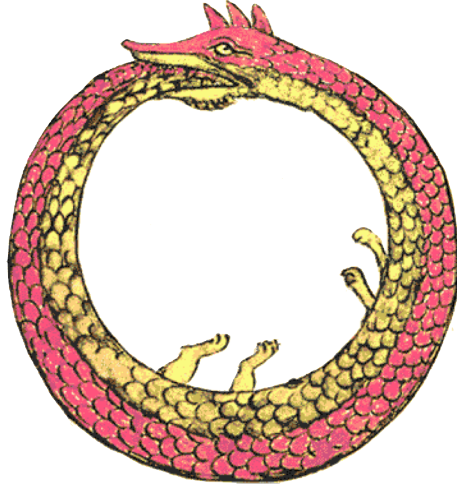
Déplace 4 disques.



Exercices



Rendez-vous sur EVAL, ici.



Algorithme :

Transférer n disques du piquet 0 (origine) vers le piquet D (destination) via le piquet I (intermédiaire):

si $n > 0$ **alors**

Transférer $n-1$ disques de 0 vers I via D

transférer le disque restant de 0 vers D

Transférer $n-1$ disques de I vers D via 0

fin si

Complexité :

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

$$T(1) = 1$$

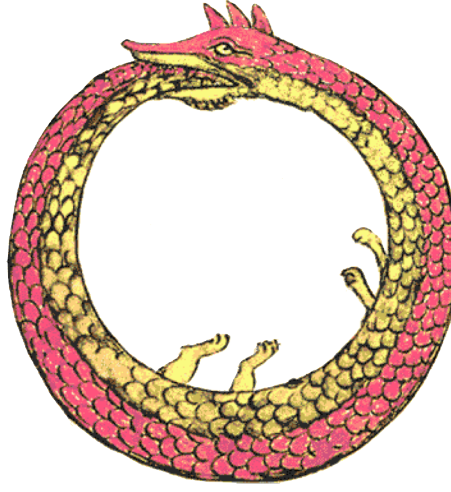
$$T(2) = 1 + T(1) + T(1)$$

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

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

2.4. Permutations



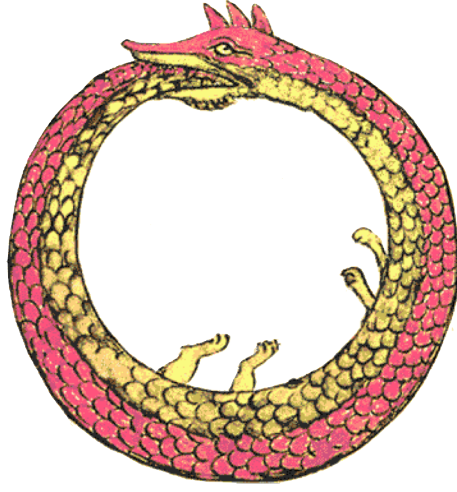


Pour faire jouer un programme au scrabble, il faut ...

- 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

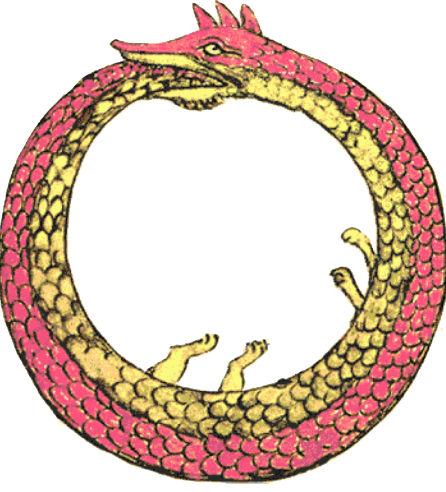
Entrée: S, par exemple ABCD
n, par exemple 3

Paramètre de
récursivité

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

- *Cas trivial* : 1 seule permutation pour une chaîne de 1 caractère
- *Cas général* : pour chacun des n premiers caractères, générer les permutations des n-1 caractères restants, suivies de ce caractère

Approche récursive



- **Cas général** : pour chacun des n premiers caractères, générer les permutations des $n-1$ caractères restants, suivies de ce caractère

Entrée : $S=ABC$, $n=3$

A B C

B A C

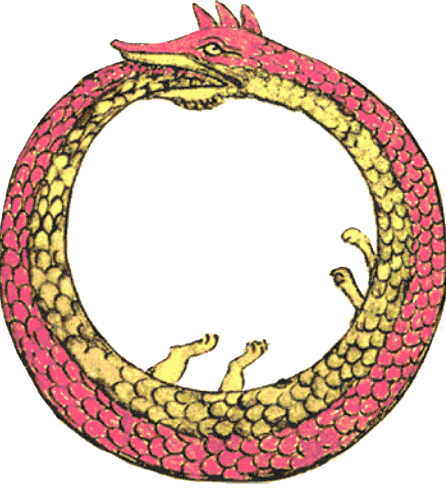
A C B

C A B

B C A

C B A

Approche récursive

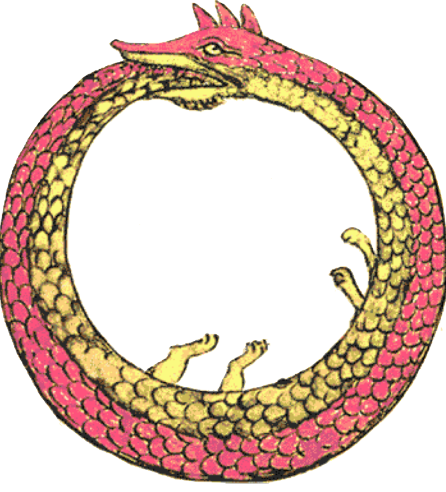


- **Cas général** : pour chacun des n premiers caractères, générer les permutations des $n-1$ caractères restants, suivies de ce caractère

Entrée : $S=ABC$, $n=3$

A	B	C	]	--C
B	A	C		
A	C	B	]	--B
C	A	B		
B	C	A	]	--A
C	B	A		

Approche récursive

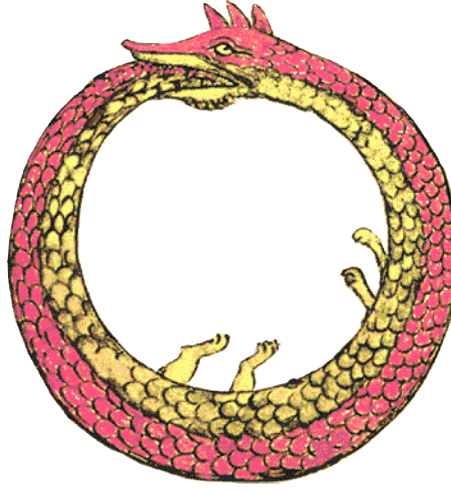


- **Cas général** : pour chacun des n premiers caractères, générer les permutations des $n-1$ caractères restants, suivies de ce caractère

Entrée : $S=ABC$, $n=3$

$$\begin{array}{l}
 perm(AB) \begin{array}{|c|c|c|} \hline A & B & C \\ \hline B & A & C \\ \hline \end{array} \left. \vphantom{\begin{array}{|c|c|c|}} \right] \begin{array}{l} \text{--}C \end{array} \\
 \\
 perm(AC) \begin{array}{|c|c|c|} \hline A & C & B \\ \hline C & A & B \\ \hline \end{array} \left. \vphantom{\begin{array}{|c|c|c|}} \right] \begin{array}{l} \text{--}B \end{array} \\
 \\
 perm(BC) \begin{array}{|c|c|c|} \hline B & C & A \\ \hline C & B & A \\ \hline \end{array} \left. \vphantom{\begin{array}{|c|c|c|}} \right] \begin{array}{l} \text{--}A \end{array}
 \end{array}$$

Approche récursive

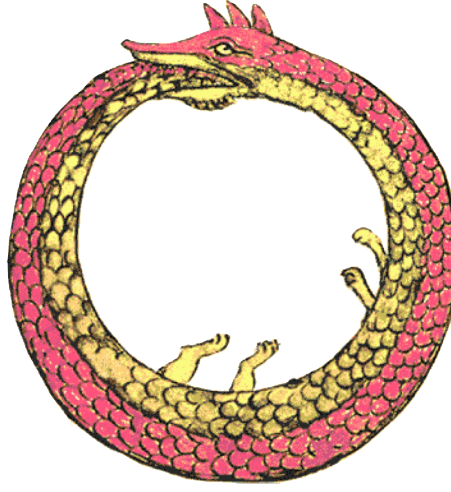


- **Cas général** : pour chacun des n premiers caractères, **le placer en position n** , puis générer les permutations des $n-1$ caractères restants.

Entrée : $S=ABCD$, $n=3$

$perm(AB)$	A B B A	C D	] --CD
		C D	
$perm(AC)$	A C C A	B D	] --BD
		B D	
$perm(BC)$	B C C B	A D	] --AD
		A D	

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 les n premières lettres c de S, boucler
```

```
      Placer c en position n
```

```
      permuter(S,n-1)
```

```
    fin boucler
```

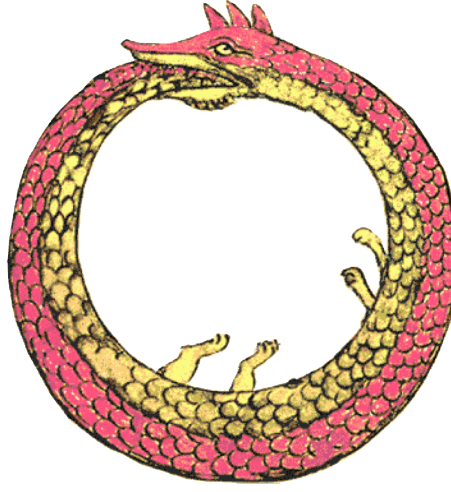
```
  fin si
```

Paramètre de
récursivité: n

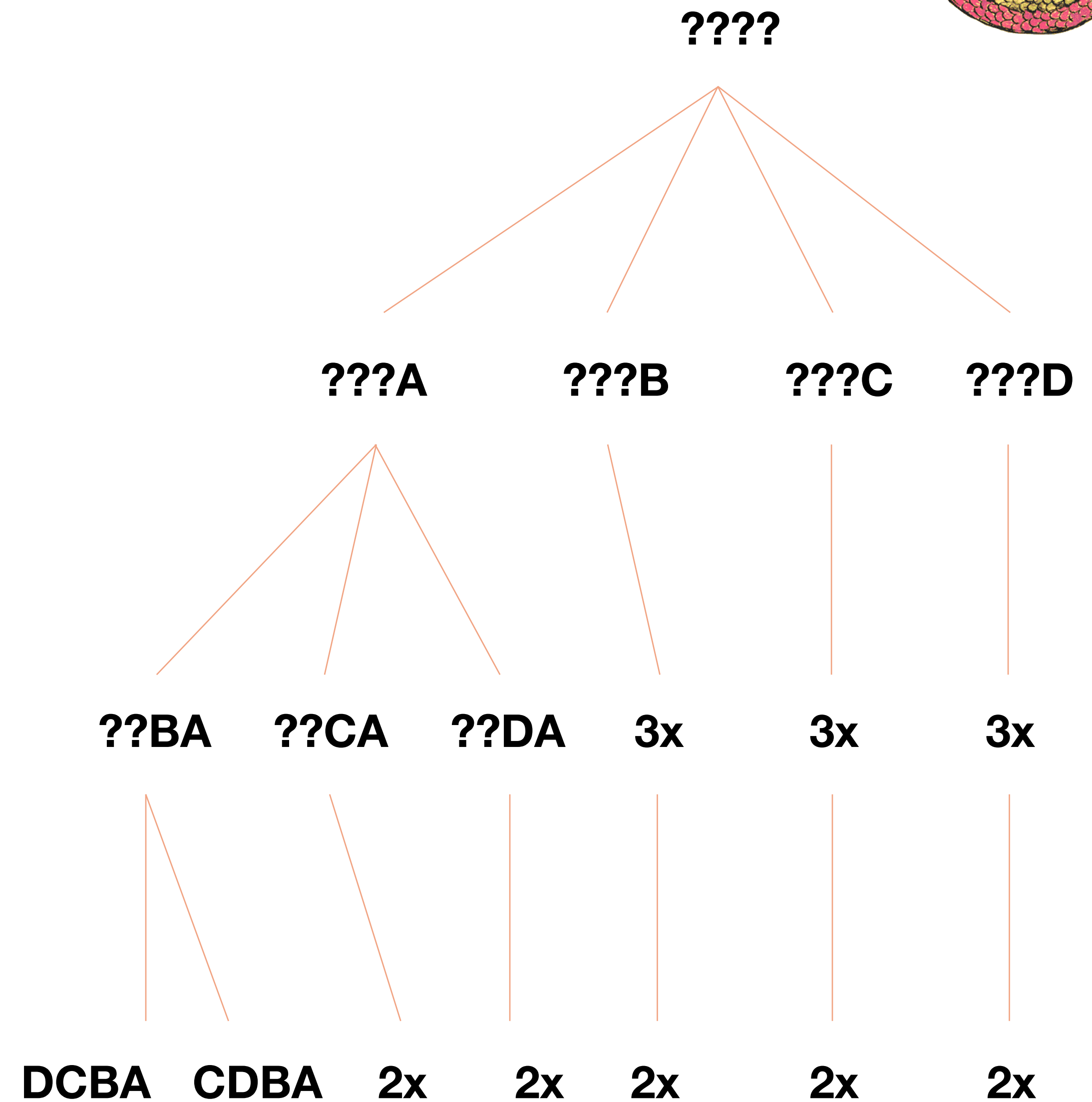
Cas trivial

Cas général

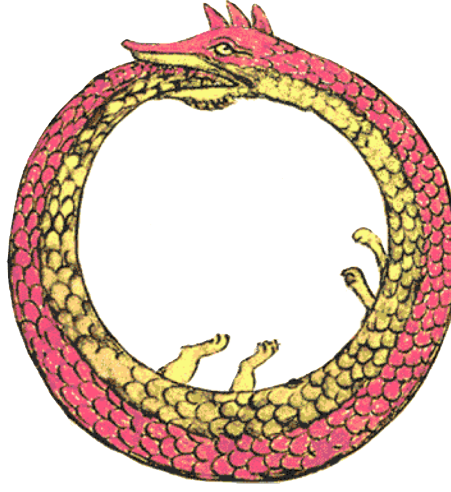
Complexité ?



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



Nombre d'appels récur­sifs



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

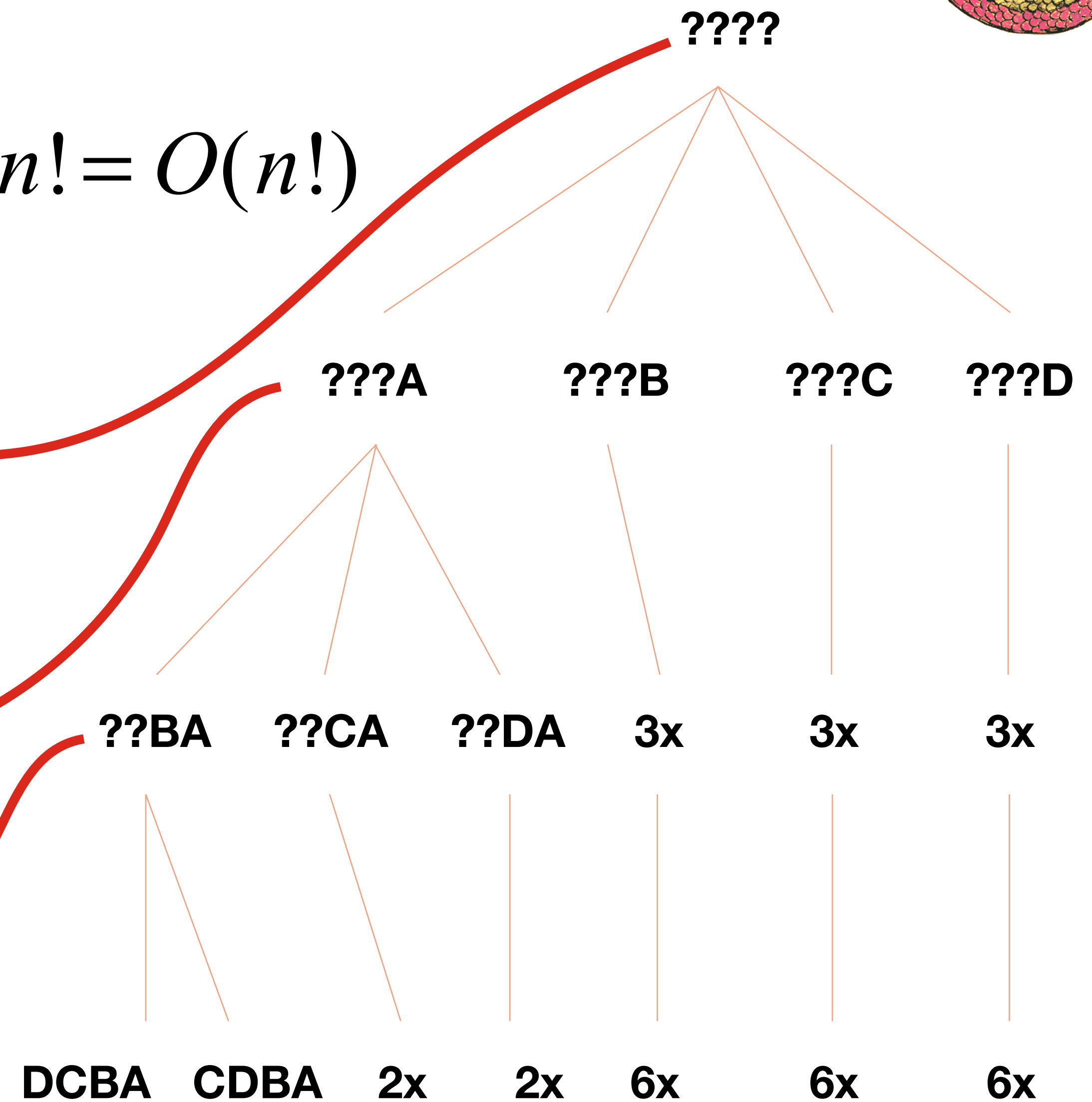
$n!/n! = 1$ appel à `permuter(S,n)`

$n!/k!$ appels à `permuter(S,k)`

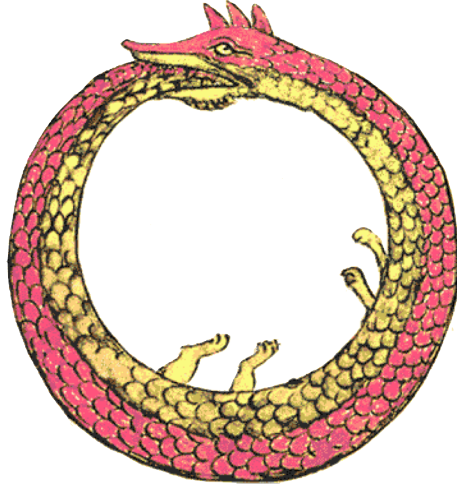
$n!/6$ appels à `permuter(S,3)`

$n!/2$ appels à `permuter(S,2)`

$n!$ appels au cas trivial



Boucler ? Placer ?



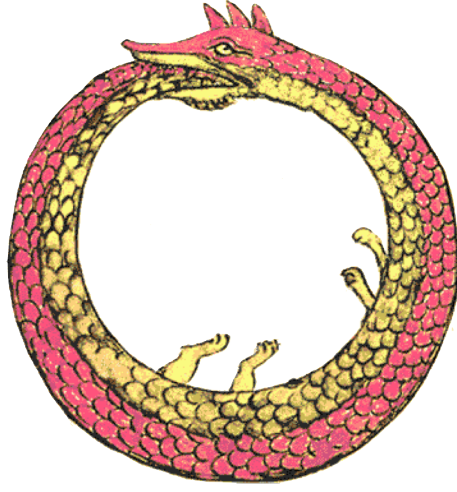
Comment ...

- ... 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 les n premières lettres c de S, boucler
      Placer c en position n
      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
      échanger S(i) et S(n)
      permuter(S,n-1)
    fin boucler
  fin si
```

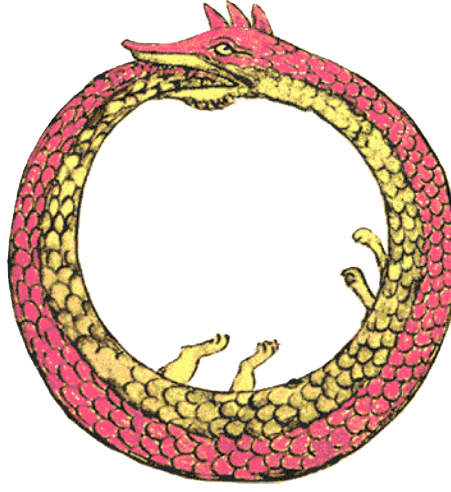
Comment passer le paramètre S de la fonction ?

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

ABC	ABC
BAC	BAC
CBA	CAB
BCA	ACB
CAB	ABC
ACB	BAC

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

Double échange

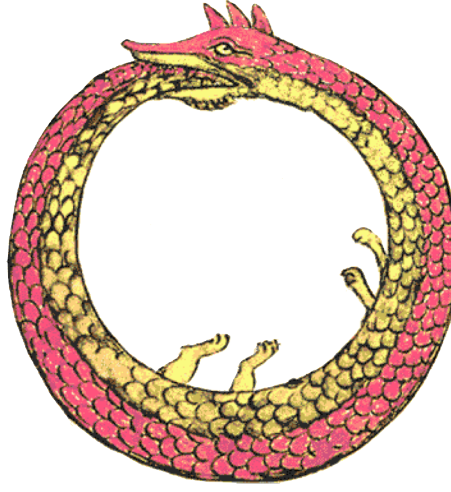


```
fonction permuter(S,n)
  si n vaut 1
    traiter(S)
  sinon
    pour i allant de 1 à n, boucler
      échanger S(i) et S(n)
      permuter(S,n-1)
      échanger S(i) et S(n)
    fin boucler
  fin si
```

- 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 $\sim 2 \cdot 1.71n! \approx 3.43 n!$

Sans échange inutile



- Echanger $S(i)$ et $S(n)$ est inutile pour $i=n$

- $2(k-1)$ échanges dans $\text{permuter}(S, k)$

$$\text{Echanges}(n) = 2 \sum_{k=2}^n (k-1) \frac{n!}{k!}$$

$$= 2 \left(\sum_{k=2}^n \frac{k \cdot n!}{k!} - \sum_{k=2}^n \frac{n!}{k!} \right)$$

$$= 2 \left(\sum_{k=2}^n \frac{n!}{(k-1)!} - \sum_{k=2}^n \frac{n!}{k!} \right)$$

$$= 2 \left(\sum_{k=1}^{n-1} \frac{n!}{k!} - \sum_{k=2}^n \frac{n!}{k!} \right)$$

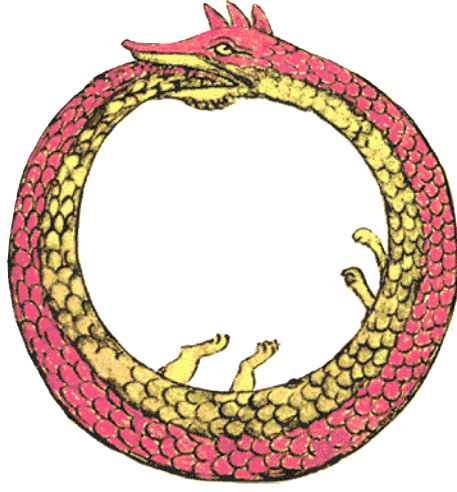
$$= 2 \left(\frac{n!}{1!} - \frac{n!}{n!} \right) = 2(n! - 1)$$

```

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

```

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: $n! - 1$