

CAML

3 - Récursivité

<http://tsi.tuxfamily.org/OCaml>



15 février 2024

Exercice

Que fait le processus ★ défini par les règles :

$$A \star |B \longrightarrow |A \star B$$

$$A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)



Exercice

Que fait le processus ★ défini par les règles :

$$\mathcal{R}_1 : A \star |B \longrightarrow |A \star B$$

$$\mathcal{R}_2 : A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)

Exercice

Que fait le processus ★ défini par les règles :

$$\mathcal{R}_1 : A \star |B \longrightarrow |A \star B$$

$$\mathcal{R}_2 : A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)

||| ★ ||

Exercice

Que fait le processus ★ défini par les règles :

$$\mathcal{R}_1 : A \star |B \longrightarrow |A \star B$$

$$\mathcal{R}_2 : A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)

$$||| \star || \xrightarrow{\mathcal{R}_1} |||| \star |$$

Exercice

Que fait le processus ★ défini par les règles :

$$\mathcal{R}_1 : A \star | B \longrightarrow | A \star B$$

$$\mathcal{R}_2 : A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)

$$||| \star || \xrightarrow{\mathcal{R}_1} |||| \star | \xrightarrow{\mathcal{R}_1} ||||| \star$$

Exercice

Que fait le processus ★ défini par les règles :

$$\mathcal{R}_1 : A \star |B \longrightarrow |A \star B$$

$$\mathcal{R}_2 : A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)

$$||| \star || \xrightarrow{\mathcal{R}_1} |||| \star | \xrightarrow{\mathcal{R}_1} ||||| \star \xrightarrow{\mathcal{R}_2} |||||$$

Exercice

Que fait le processus \star défini par les règles :

$$\mathcal{R}_1 : A \star |B \longrightarrow |A \star B$$

$$\mathcal{R}_2 : A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)

$$||| \star || \xrightarrow{\mathcal{R}_1} |||| \star | \xrightarrow{\mathcal{R}_1} ||||| \star \xrightarrow{\mathcal{R}_2} |||||$$

On vient de définir l'addition !

Exercice

$$A \star | B \longrightarrow | A \star B$$

$$A \star \longrightarrow A$$

où A et B représentent des motifs quelconques, et $|$ une unité (par exemple $5 = |||||$)

```
let rec etoile a = function
  | 0 -> a
  | b -> etoile (1+a) (b-1)
;;
```

Une nouvelle façon de penser
Exemple de n !
Suite de Fibonacci
A quoi sert la récursivité ?
Le problème de terminaison

Un exemple
Déclaration



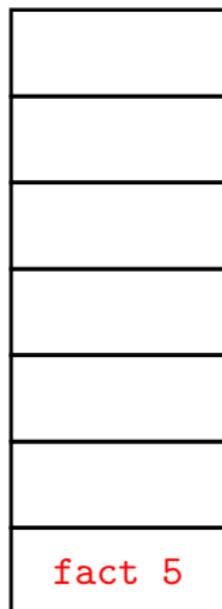
Déclaration de la fonction factorielle, basée sur le fait que

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

```
let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;
```



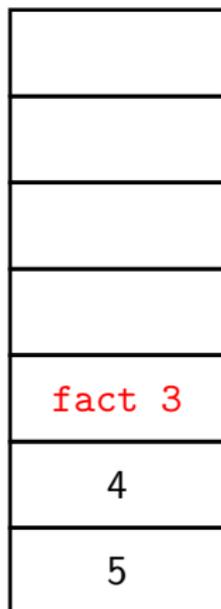
- Empilement des données.



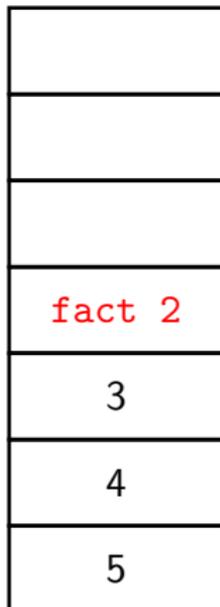
- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$



- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$



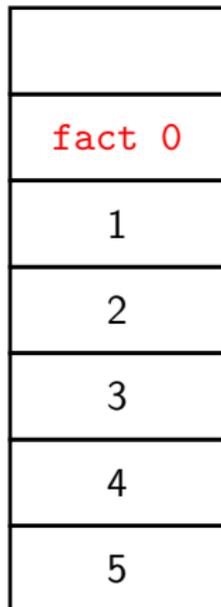
- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$



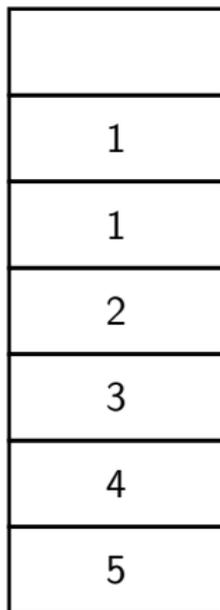
- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$



- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$



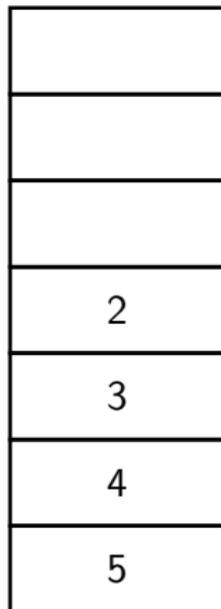
- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times 1.$



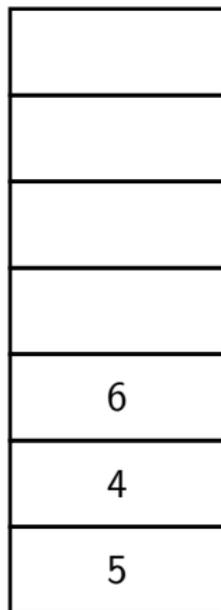
- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times 1.$
- Dépilement des données.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1.$



- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times 1.$
- Dépilement des données.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1.$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2.$



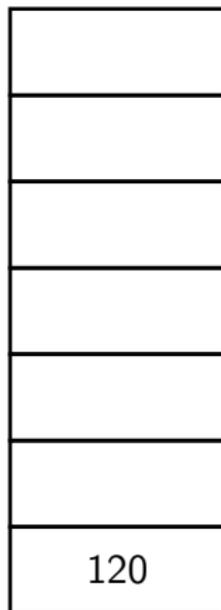
- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \mathbf{1}$.
- Dépilement des données.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \mathbf{1}$.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \mathbf{2}$.
 - $\text{fact } 5 = 5 \times 4 \times \mathbf{6}$.



- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \mathbf{1}$.
- Dépilement des données.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \mathbf{1}$.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \mathbf{2}$.
 - $\text{fact } 5 = 5 \times 4 \times \mathbf{6}$.
 - $\text{fact } 5 = 5 \times \mathbf{24}$.



- Empilement des données.
 - $\text{fact } 5 = 5 \times \text{fact } 4$
 - $\text{fact } 5 = 5 \times 4 \times \text{fact } 3$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \text{fact } 2$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \text{fact } 1$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times 1 \times \mathbf{1}$.
- Dépilement des données.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times 2 \times \mathbf{1}$.
 - $\text{fact } 5 = 5 \times 4 \times 3 \times \mathbf{2}$.
 - $\text{fact } 5 = 5 \times 4 \times \mathbf{6}$.
 - $\text{fact } 5 = 5 \times \mathbf{24}$.
 - $\text{fact } 5 = \mathbf{120}$.



```

let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;

#trace fact;;

fact 5;;

#untrace fact;;

fact 5;;

```

```

# #trace fact;;
fact is now traced.
# fact 5;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
- : int = 120
# #untrace fact;;
fact is no longer traced.
# fact 5;;
- : int = 120

```

Pourquoi cet exemple n'est pas forcément pertinent?

```
let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;
```

Par rapport à la version impérative :

- Coût en temps?

Pourquoi cet exemple n'est pas forcément pertinent?

```
let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;
```

Par rapport à la version impérative :

- Coût en temps?
- Coût en mémoire?

Pourquoi cet exemple n'est pas forcément pertinent?

```
let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;
```

Par rapport à la version impérative :

- Coût en temps ?
- Coût en mémoire ?
- Une version en temps constant ?

Pourquoi cet exemple n'est pas forcément pertinent ?

```
let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;
```

Par rapport à la version impérative :

- Coût en temps ?
- Coût en mémoire ?
- Une version en temps constant ?

```
let fact n =
  let t = [1;1;2;6;24;120;720;5040;40320;362880;
          3628800; ..... ;2432902008176640000] in t.(n)
;;
```

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1



fact2

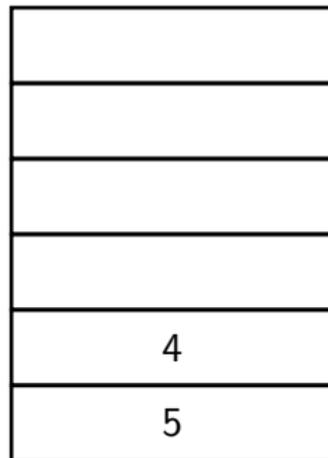
(1, fact2 5)

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1



fact2

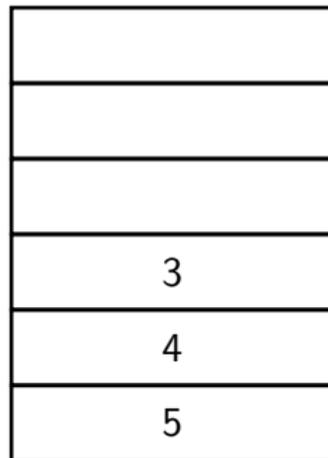
(5, fact2 4)

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1



fact2

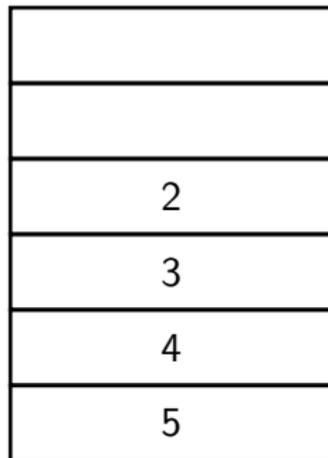
(20, fact2 3)

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1



fact2

(60, fact2 2)

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1

1
2
3
4
5

fact2

(120, fact2 1)

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1

1
1
2
3
4
5

fact2

(120, fact2 0)

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1

1
2
3
4
5

fact2

→ 120

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1

2
3
4
5

fact2

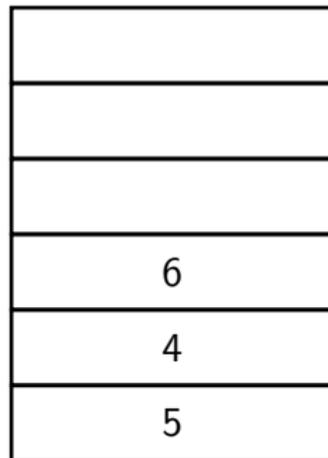
→ 120

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1



fact2

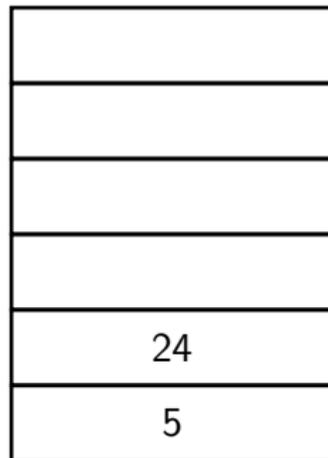
→ 120

Quelle est la différence entre ces 2 programmes?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1



fact2

→ 120

Quelle est la différence entre ces 2 programmes?

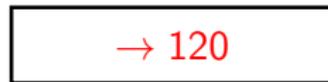
```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
in fact1 5
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
in fact2 1 5
;;
```

fact1



fact2



Quelle est la différence entre ces 2 programmes ?

```
let rec fact1 = function
  | 0 -> 1
  | n -> n*fact1 (n-1)
;;
```

```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
;;
```

Vocabulaire

La fonction fact2 est dite **récursive terminale**, la fonction fact1 est dite **récursive non terminale**

$$f(x) = \text{si } R(x) \text{ alors } g(x) \text{ sinon } h(x, f(k(x)))$$

est la définition d'une fonction **récursive non terminale** : si la condition $R(x)$ est remplie, alors le calcul se termine avec l'évaluation de la fonction g . Dans le cas contraire, on effectue un appel récursif à f qui, lorsqu'il sera terminé, n'achèvera pas le calcul car il faudra encore évaluer $h(x, \dots)$. Chaque appel à f est associé à une zone en mémoire où l'ensemble du contexte (à savoir les paramètres qui ont été passés à f ainsi que l'endroit où l'appel s'est interrompu) est conservé : chacun des appels récursifs peut ainsi se terminer.

$$f(x) = \text{si } R(x) \text{ alors } g(x) \text{ sinon } f(k(x))$$

est la définition d'une fonction **récursive terminale**. Lorsque l'appel $f(k(x))$ se termine, alors l'appel précédent se termine aussi.

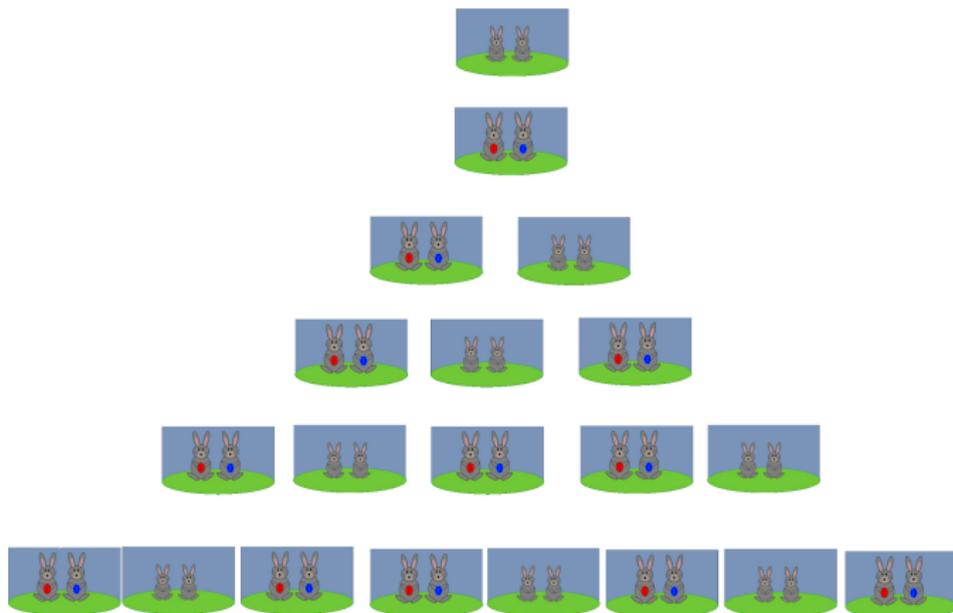
```
let rec fact2 c = function
  | 0 -> c
  | n -> fact2 (c*n) (n-1)
;;
```

L'inconvénient avec `fact2` c'est que pour avoir $n!$, on doit faire l'appel avec `fact2 1 n`. On procède alors comme ceci à l'aide d'une fonction auxiliaire :

```
let fact2 n =
  let rec factR c = function
    | 0 -> c
    | n -> factR (c*n) (n-1)
  in factR 1 n;;
```

Lorsque le compilateur la gère, la programmation d'une fonction récursive terminale a principalement pour objectif de limiter la complexité spatiale d'une fonction

Dans notre exemple, en réalité ces différents algorithmes sont à peu près aussi efficaces puisque avec la limitation sur les entiers, $21!$ dépasse les limites et affiche un résultat négatif.



$$F_0 = F_1 = 1 \text{ et } \forall n \in \mathbb{N}^*, F_{n+1} = F_n + F_{n-1}$$

Exercice

Écrire un programme récursif naïf utilisant simplement la définition qui calcule le $n^{\text{ème}}$ terme.



```
let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo(n-1) + fibo(n-2)
;;
```

```
let rec fibo = fonction
  | 0 | 1 -> 1
  | n -> fibo(n-1) + fibo(n-2)
;;
```

Exercice

Que dire de ce programme ? Donner un équivalent du nombre d'appels nécessaires à calculer `fibo n` lorsque n est très grand.



Aide : Noter C_n le nombre d'appels à la fonction `fibo` à réaliser pour calculer F_n et trouver une relation de récurrence entre C_{n+1} et C_n ...

```
let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;
```

Notons C_n le nombre d'appels à réaliser pour calculer F_n .

```
let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;
```

Notons C_n le nombre d'appels à réaliser pour calculer F_n .

- $C_0 = C_1 = 1$
- Dans les autres cas, $C_{n+1} = 1 + C_n + C_{n-1}$

```
let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;
```

Notons C_n le nombre d'appels à réaliser pour calculer F_n .

- $C_0 = C_1 = 1$
- Dans les autres cas, $C_{n+1} = 1 + C_n + C_{n-1}^{(*)}$

(*) peut s'écrire $C_{n+1} + 1 = C_n + 1 + C_{n-1} + 1$

```

let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;

```

Notons C_n le nombre d'appels à réaliser pour calculer F_n .

- $C_0 = C_1 = 1$
- Dans les autres cas, $C_{n+1} = 1 + C_n + C_{n-1}^{(*)}$

(*) peut s'écrire $C_{n+1} + 1 = C_n + 1 + C_{n-1} + 1$

En posant $u_n = C_n + 1$, on obtient

$$\begin{cases} u_0 = u_1 = 2 \\ u_{n+1} = u_n + u_{n-1} \forall n \in \mathbb{N}^* \end{cases}$$

avec $u_n = C_n + 1$, $\begin{cases} u_0 = u_1 = 2 \\ u_{n+1} = u_n + u_{n-1} \forall n \in \mathbb{N}^* \end{cases}$

Il s'agit d'une suite linéaire récurrente d'ordre 2, que l'on sait résoudre....



avec $u_n = C_n + 1$, $\begin{cases} u_0 = u_1 = 2 \\ u_{n+1} = u_n + u_{n-1} \forall n \in \mathbb{N}^* \end{cases}$

$$u_n = \frac{5 - \sqrt{5}}{5} \left(\frac{1 - \sqrt{5}}{2} \right)^n + \frac{5 + \sqrt{5}}{5} \left(\frac{1 + \sqrt{5}}{2} \right)^n$$

$$\text{avec } u_n = C_n + 1, \begin{cases} u_0 = u_1 = 2 \\ u_{n+1} = u_n + u_{n-1} \forall n \in \mathbb{N}^* \end{cases}$$

$$u_n = \frac{5 - \sqrt{5}}{5} \underbrace{\left(\frac{1 - \sqrt{5}}{2} \right)^n}_{\approx -0,6} + \frac{5 + \sqrt{5}}{5} \underbrace{\left(\frac{1 + \sqrt{5}}{2} \right)^n}_{\approx 1,6}$$

$$\text{avec } u_n = C_n + 1, \begin{cases} u_0 = u_1 = 2 \\ u_{n+1} = u_n + u_{n-1} \forall n \in \mathbb{N}^* \end{cases}$$

$$u_n = \frac{5 - \sqrt{5}}{5} \underbrace{\left(\frac{1 - \sqrt{5}}{2}\right)^n}_{\approx -0,6} + \frac{5 + \sqrt{5}}{5} \underbrace{\left(\frac{1 + \sqrt{5}}{2}\right)^n}_{\approx 1,6}$$

$$C_n \sim u_n = O\left(\frac{1 + \sqrt{5}}{2}\right)^n$$

Exercice

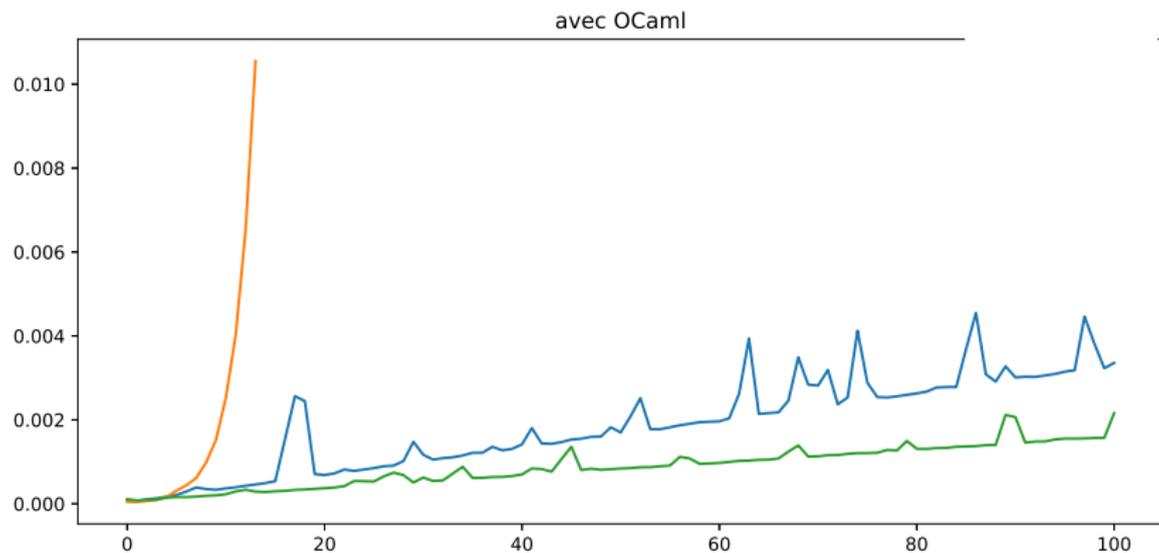
Améliorer l'idée précédente en une fonction récursive terminale.



```
let fibo n =  
  let rec fiboR a b n = match n with  
    | 0 -> a  
    | n -> fiboR b (a+b) (n-1)  
  in fiboR 1 1 n  
;;
```

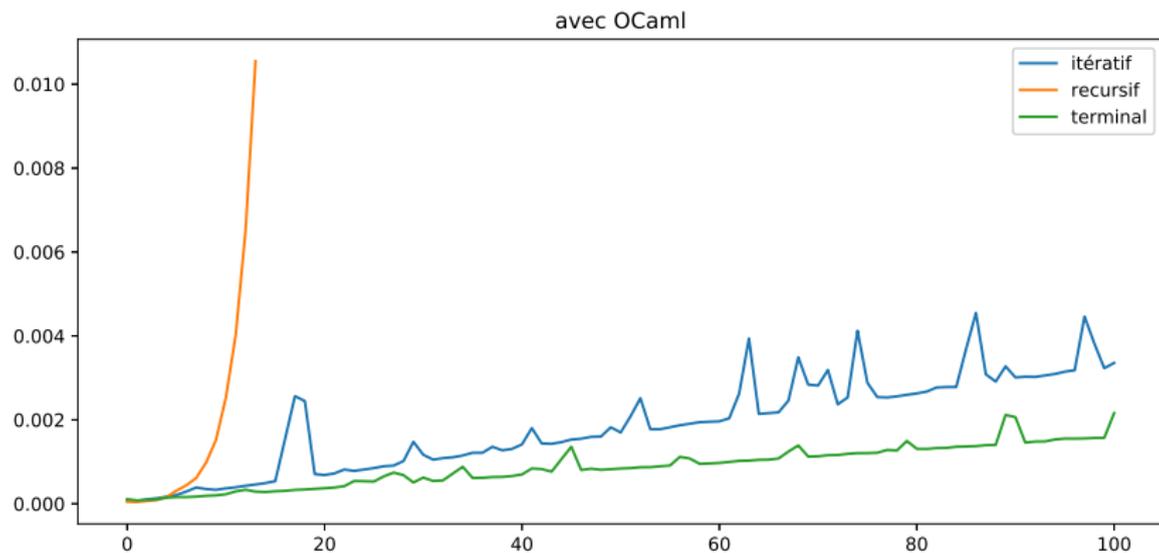
ou

```
let fibo n =  
  let rec fiboR a b = function  
    | 0 -> a  
    | n -> fiboR b (a+b) (n-1)  
  in fiboR 1 1 n  
;;
```



Exercice

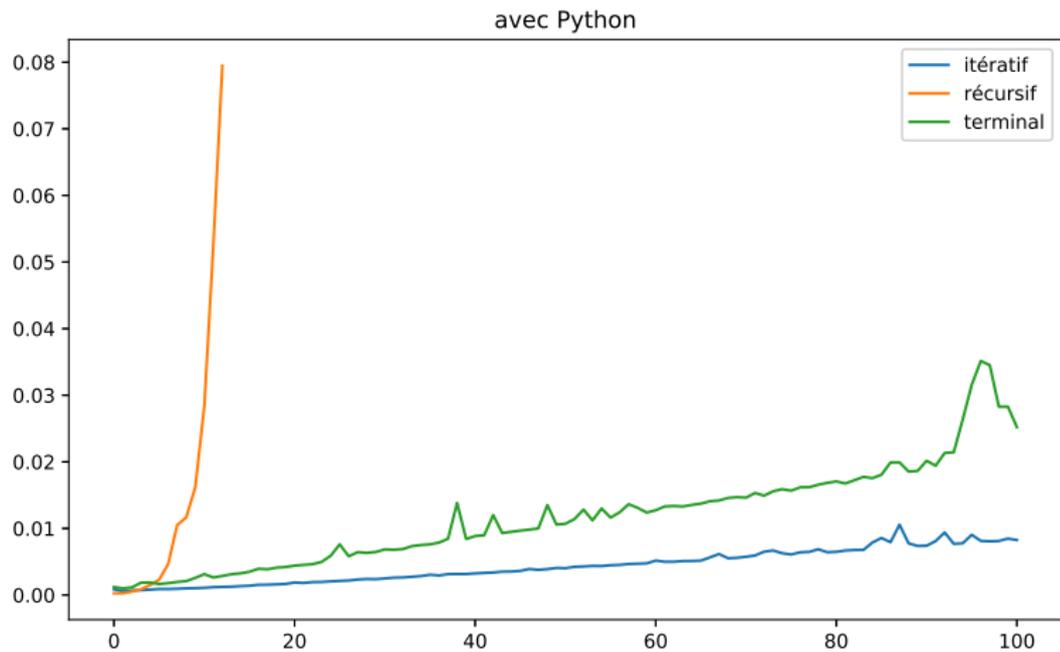
Quelles couleurs pour quels algorithmes ?



Exercice

Quelles couleurs pour quels algorithmes ?

```
def fibo1(n) :  
    moins_un = 1  
    moins_deux = 1  
    u_n = 1  
    for i in range(2,n+1) :  
        u_n = moins_un + moins_deux  
        moins_deux = moins_un  
        moins_un = u_n  
    return u_n  
  
def fibo2(n) :  
    return 1 if n < 2 else fibo2(n-1) + fibo2(n-2)  
  
def fibo3(n) :  
    def fiboR(a,b,n) :  
        return a if n==0 else fiboR(b, a+b, n-1)  
    return fiboR(1, 1, n)
```



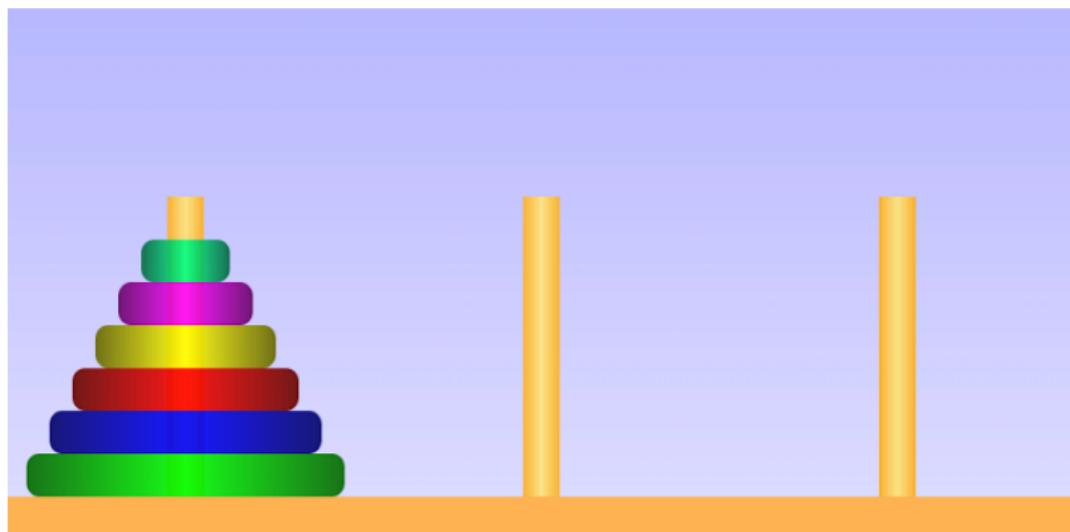
Quel est l'intérêt d'une fonction récursive, si c'est pour faire juste aussi bien que des fonctions itératives ?

```
let rec print n =  
    print_int n;  
    if n > 0 then print (n-1);;  
  
let rec print n =  
    if n > 0 then print (n-1);  
    print_int n;;
```

Exercice

Quelles différences entre ces 2 fonctions ?

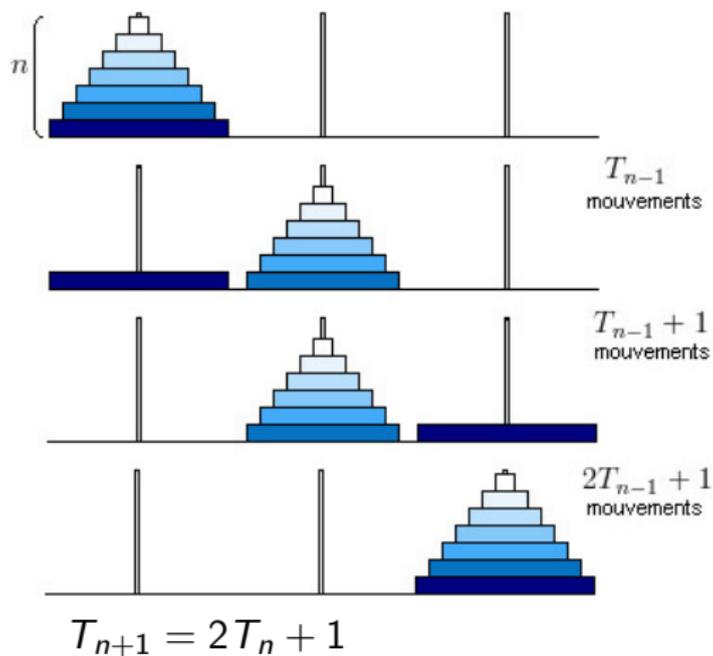


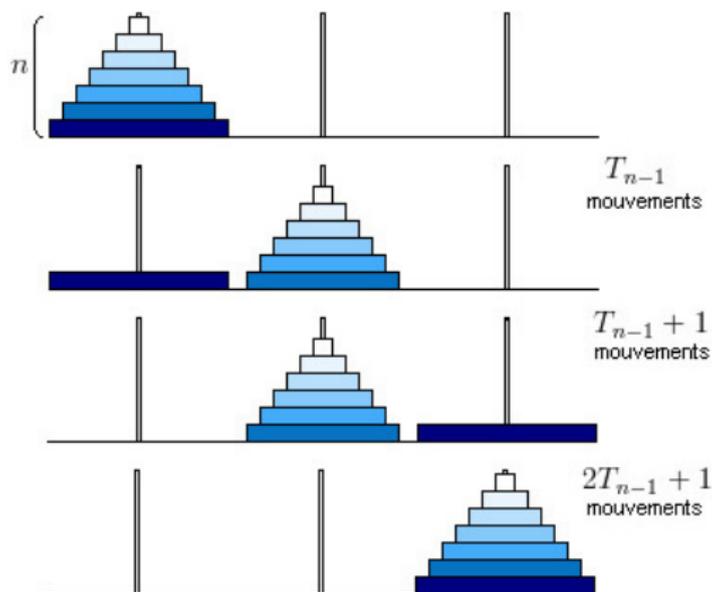


Exercice

Résoudre le problème avec 1, 2 puis 3 disques.
Pour une tour de n étages, quel est le nombre minimum de déplacements à réaliser ?







$$T_{n+1} = 2T_n + 1 \text{ puis } T_n = 2^n - 1$$

```
let rec hanoi n init inter final =  
  if n > 0  
  then begin  
    hanoi (n-1) init final inter;  
    print_string ("Déplacer un disque de " ^ init  
                  ^ " vers " ^ final ^ "\n");  
    hanoi (n-1) inter init final  
  end;  
;;
```

```
let rec hanoi n init inter final =  
  if n > 0  
  then begin  
    hanoi (n-1) init final inter;  
    print_string ("Déplacer un disque de "^init  
                  ^" vers "^final^"\n");  
    hanoi (n-1) inter init final  
  end;  
;;
```

Bon courage à celles et ceux qui veulent réaliser ce programme en version non récursive...

Un exemple qui se termine

```
let rec u = function
  | 0 ->  $\alpha$ 
  | n ->  $\mathcal{F}(u(n-1))$ 
;;
```

où

- α est une valeur de type 'a
- \mathcal{F} est une fonction de type 'a -> 'a

Cette fonction rend le $n^{\text{ième}}$ terme de la suite (u_n) définie par

$$\begin{cases} u_0 = \alpha \\ u_{n+1} = \mathcal{F}(u_n), \forall n \in \mathbb{N} \end{cases}$$

Un exemple qui ne se termine pas : la fonction de Morris

```
let rec m p q = match p with
  | 0 -> 1
  | _ -> m (p-1) (m p q)
;;
```

$m(1, 0) = m(0, m(1, 0))$ pourtant le calcul ne termine pas : le passage d'argument se fait par **valeur**, c'est à dire que la valeur des arguments est évaluée **avant** que la fonction ne le soit.

Un exemple qui se termine peut-être... :
la fonction **Q de Hofstadter**

```
let rec q = function
  | 1 -> 1
  | 2 -> 1
  | n -> q(n - q(n-1)) + q(n - q(n-2))
;;
```

Ni la terminaison, ni la non terminaison de cette fonction ne sont démontrées, c'est un problème encore ouvert à ce jour.

Problème de l'arrêt

Il n'existe pas de moyen algorithmique de déterminer la terminaison de n'importe quelle fonction.

Cette affirmation est mathématiquement démontrable : on dit que le problème d'arrêt est un problème indécidable.

Principe de récurrence simple

Soit \mathcal{P} une propriété définie sur \mathbb{N} telle que

$$\begin{cases} \mathcal{P}(0) \text{ est vraie} \\ \forall n \in \mathbb{N}, \mathcal{P}(n) \implies \mathcal{P}(n+1) \end{cases}$$

alors $\forall n \in \mathbb{N}, \mathcal{P}(n)$ est vraie.

Ensemble ordonné bien fondé

Un ensemble ordonné $(E; \preceq)$ est bien fondé lorsque toute partie non vide possède un élément minimal.

Principe d'induction

Soit $(E; \preceq)$ un ensemble ordonné bien fondé, A une partie non vide de E et $\varphi : E \setminus A \rightarrow E$ telle que $\forall x \in E \setminus A, \varphi(x) \prec x$. Soit \mathcal{P} une propriété définie sur E telle que

$$\begin{cases} \forall a \in A, \mathcal{P}(a) \text{ est vraie} \\ \forall x \in E \setminus A, \mathcal{P}(\varphi(x)) \implies \mathcal{P}(x) \end{cases}$$

alors $\forall x \in E, \mathcal{P}(x)$ est vraie

Le principe d'induction permet de justifier la terminaison des fonctions f du type :

Propriété

$$f : E \longrightarrow F$$
$$x \longmapsto \begin{cases} g(x), & \text{si } x \in A \\ h(x, f(\varphi(x))), & \text{sinon} \end{cases}$$

avec

- $g : A \longrightarrow F$,
- $h : (E \setminus A) \times F \longrightarrow F$
- $\varphi : (E \setminus A) \longrightarrow E$ telle que $\forall x \in E \setminus A, \varphi(x) \prec x$.

factorielle

```
let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;
```

factorielle

```
let rec fact = function
  | 0 -> 1
  | n -> n*fact (n-1)
;;
```

$E = \mathbb{N}$ muni de l'ordre lexicographique
 $A = \{0\}$; $\varphi : n \mapsto n - 1$

PGCD

```
let rec pgcd a b = match a with  
  | 0 -> b  
  | b -> pgcd (a mod b) b  
;;
```

PGCD

```
let rec pgcd a b = match a with  
  | 0 -> b  
  | b -> pgcd (a mod b) b  
;;
```

$$E = \mathbb{N}^2 ; A = \{0\} \times \mathbb{N} ; \varphi : (p, q) \mapsto (q \bmod p, p)$$

Relation d'ordre : ordre lexicographique.

Fibonacci

```
let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;
```

Fibonacci

```
let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;
```

$E = \mathbb{N}$; $A = \{0, 1\}$; $\varphi_1 : n \mapsto n - 1$ et $\varphi_2 : n \mapsto n - 2$

Sources :

- <http://recursivite.developpez.com/>
- <http://www.pas-a-pas.eu/vosarticles.php?id=8>
- <http://abstractstrategygames.blogspot.fr/2010/11/towers-of-hanoi.html>
- De nombreux cours sur internet de CPGE...