# 1 Récursion

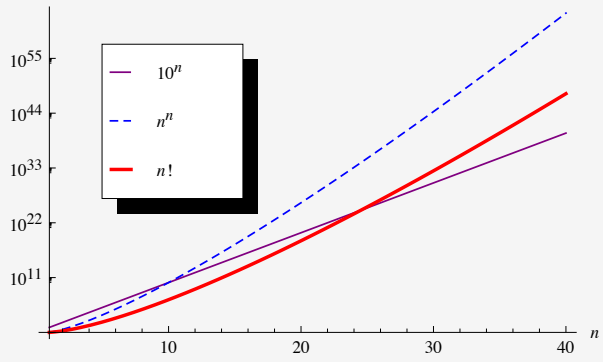
## 1.1 Factorielle

Définition de la factorielle : $0! = 1$ et $n! = 1 \times 2 \times 3 \times \cdots \times n = \prod_{k=1}^{n} k$. pour $n > 0$. W(fr)

**Définition 1.1.** *On définit la* factorielle $n!$ *d'un nombre naturel* $n \in \{0, 1, 2, 3 \dots\}$ *par*

$$n! = \begin{cases} 1 & \{n = 0\} \\ n \cdot (n-1)! & \{n > 0\} \end{cases}$$

## 1.2 Croissance de la factorielle et la formule de Stirling




La factorielle croît très rapidement — c'est une fonction **superexponentielle** : pour tout $c > 1$ fixe, il existe un $n_0(c)$ t.q.

$$c^n < n! \qquad \{n = n_0(c), n_0(c)+1, \dots\} \quad (1.1)$$

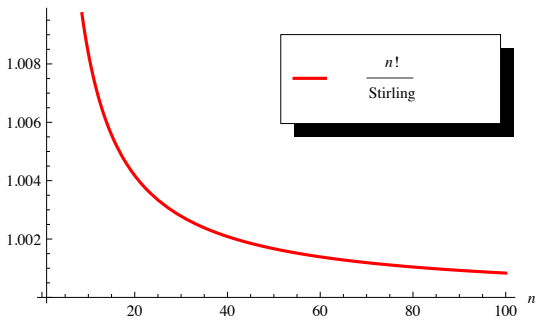
Par exemple, avec $c = 10$, $n_0(c) = 25$ suffit : $25! = 15511210043330985984000000 > 10^{25}$.

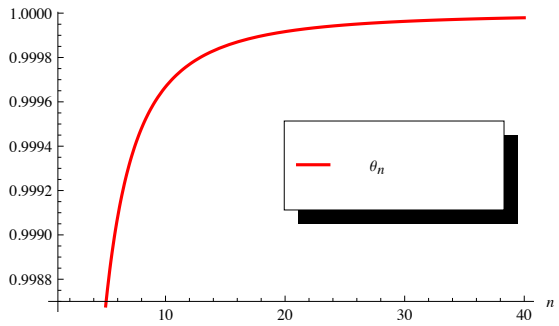
W(fr)

**Théorème 1.1** (Formule de Stirling)**.** *Pour tout* $n = 1, 2, \dots$ *il existe* $0 < \theta_n < 1$ *t.q.*

$$n! = \underbrace{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}_{\text{formule de Stirling}} \times \underbrace{\exp\left(\frac{\theta_n}{12n}\right)}_{\text{erreur de l'ordre } 1/n} .$$

Donc, $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. La formule de Stirling donne une borne inférieure serrée (et donc (1.1) est correcte avec $n_0(c) = \lceil ce \rceil$) :



**Exercice 1.1.** *Utiliser la formule de Stirling pour caractériser la croissance asymptitique de la* factorielle double *:*

$$(2k+1)!! = 1 \cdot 3 \cdot 5 \cdot 7 \cdots (2k+1) \quad (2k)!! = 2 \cdot 4 \cdot 6 \cdots (2k) \qquad \{k = 0, 1, 2, \dots\}$$

***Indice :*** *écrire* $(2k)!! = (k!) \prod_{i=1}^{k} 2$, $(2k+1)!! = \frac{(2k+1)!}{\prod_{i=1}^{k}(2i)} = \frac{(2k+1)!}{(k!)\prod_{i=1}^{k} 2}$, *et se servir de la formule de Stirling.*

## 1.3 Pile d'exécution

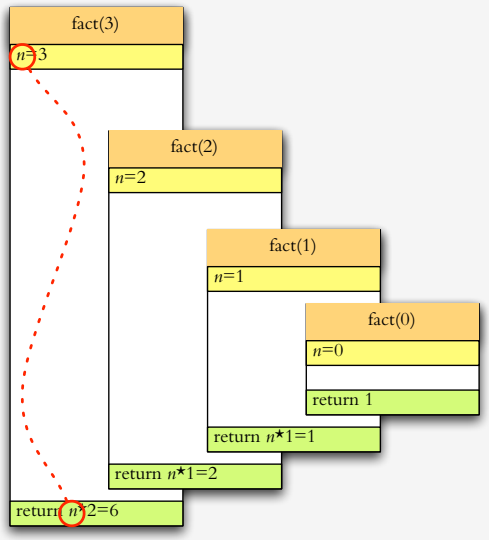
W(fr)

Définition 1.1 se traduit en un **algorithme récursif** :

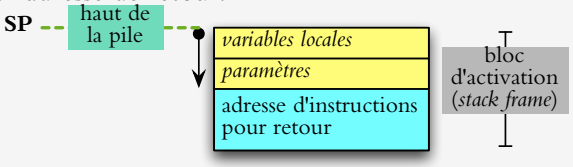
```
    FACT(n)                                   //(calcul de n!)
F1  if n = 0 then return 1
F2  else return n × FACT(n − 1)               // appel récursif
```

```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

On peut vérifier que l'algorithme satisfait les règles minimales pour récursion : (1) il y a un **cas terminal**, et (2) chaque appel récursif nous rend «plus proche» à un cas terminal. En conséquence, l'algorithme finit en un nombre fini d'appels récursifs pour tout $n$.




Lors de l'exécution, il faut recupérer le contexte (p.e., la valeur du paramètre $n$) après le retour de l'appel récursif.
Il est impossible de prédire la profondeur de la récursion au temps de compilation : on doit allouer le mémoire pour les variables locales lors de l'exécution. Pour cela, on utilise une **pile d'exécution**. Lors de l'activation d'une procédure, un bloc est alloué sur la pile pour stocker les paramètres, les variables locales et l'adresse de retour.




```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

Dans le code machine, SP dénote le pointeur de pile (*stack pointer*). Un bloc d'activation contient deux emplacements en mémoire : $n$ est stocké à l'adresse SP − 1, et l'adresse de retour se trouve à SP − 2. $r_0$ et $r_1$ sont des registres du CPU. Registre $r_0$ est utilisé pour retourner une valeur.

```
    code machine imaginaire pour fact
M1  load r1, [SP − 1]                    // (r1 contient n)
M2  jumpnonzero r1, M6
M3  load r0, 1                           // (valeur retournée en r0)
M4  sub SP, 2                            // (rétablissement de la pile)
M5  jump [SP]
M6  sub r1, 1                            // (r1 = n − 1)
M7  store M11, [SP]                      // (adresse de retour)
M8  add SP, 2                            // (allocation de frame)
M9  store r1, [SP − 1]                   // (argument de l'appel)
M10 jump M1                              // (exécution de l'appel)
M11 mul r0, [SP − 1]                     // (valeur retournée en r0)
M12 sub SP, 2                            // (rétablissement de la pile)
M13 jump [SP]
```

**Variables.** variable = abstraction d'un emplacement en mémoire [John von Neumann]
nom + adresse (lvalue) + valeur (rvalue) + type + portée

Variable locale (paramètres de fonction inclus) : adresse est rélatif au bloc d'activation ⇒ chaque copie activée de la fonction possède ses propres variables locales.

## 1.4 Nombres Fibonacci

**Définition 1.2.** *On définit les* ***nombres Fibonacci*** $F(n)$ *pour* $n = 0, 1, 2, \ldots$ *:* W(fr)

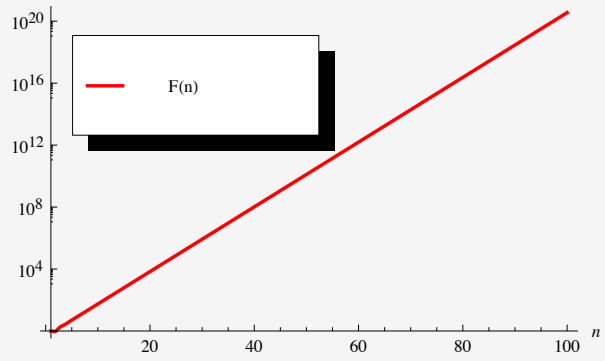
$$F(0) = 0; \quad F(1) = 1; \quad F(n) = F(n-1) + F(n-2) \qquad \{n > 1\} \tag{1.2}$$

**Formule de Binet.** Les racines de l'équation de récurrence homogène [$x^n = x^{n-1} + x^{n-2}$] sont $\phi = \frac{1+\sqrt{5}}{2} = 1.618\cdots$ et $\phi_- = 1 - \phi = \frac{1-\sqrt{5}}{2} = -0.618\cdots$. La solution spécifique se trouve par la solution des équations

$$F(0) = 0 = a\phi^0 + b\phi_-^0; \qquad F(1) = 1 = a\phi^1 + b\phi_-^1.$$

On obtient $a = -b = 5^{-1/2}$, donc

$$F(n) = \frac{\phi^n - \phi_-^n}{\sqrt{5}}. \tag{1.3}$$




Dans d'autres mots, $F(n)$ a une croissance exponentielle. Comme $\left|\phi_-^n/\sqrt{5}\right| < 1/2$ pour tout $n \geq 0$ et $F(n)$ est entier, on voit aussi que $F(n)$ égale à $\phi^n/\sqrt{5}$, arrondi au plus proche entier :

$$F(n) = \left\lfloor \phi^n/\sqrt{5} + 1/2 \right\rfloor.$$

**Exercice 1.2.** *Caractériser la croissance des nombres Pell* $P(n)$ *pour* $n = 0, 1, 2, 3, \ldots$ *:*

$$P(0) = 0; P(1) = 1; P(n) = 2P(n-1) + P(n-2) \qquad \{n > 1\}.$$

## 1.5 Algorithme d'Euclide

W(fr)

L'algorithme d'Euclide trouve le plus grand commun diviseur de deux entiers positifs.

GCD$(a, b)$ // $\{b \leq a\}$
E1 **if** $b = 0$ **then return** $a$
E2 **else return** GCD$\big(b, a \bmod b\big)$ ;

```
int gcd(int a, int b)
{   assert (b<=a && b>=0);
    if (b==0) return a;
    else return gcd(b, a%b);
}
```

Le théorème suivant avec Eq. (1.3) montre que l'algorithme d'Euclide prend un temps logarithmique en $b$.

**Théorème 1.2.** *Soit $n$ le plus grand entier pour lequel $F(n) \leq b < F(n+1)$ dans l'appel à l'algorithme d'Euclide. Alors, l'algorithme exécute au plus $(n-1)$ récursions.*

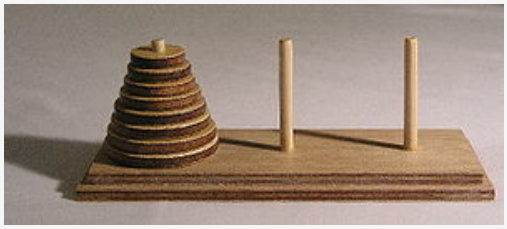
*Démonstration.* On définit $n_i$ pour $i = 1, 2, \ldots$ comme l'indice du nombre Fibonacci pour lequel $F(n_i) \leq a < F(n_i + 1)$ au début de récursion $i$ ($i = 1$ dans l'appel initiel). Si $b < F(n_i)$, on a immédiatement $n_{i+1} \leq n_i - 1$ dans l'appel suivant. Si $b \geq F(n_i)$, alors $a \bmod b \leq a - b < F(n_i - 2)$. Donc, après 2 appels, on a $a < F(n_i - 2)$ et $n_{i+2} \leq n_i - 2$. En conséquence, le nombre d'appels est borné par $n_1 - 2$ : avec $n_i \leq 3$, on a $0 \leq b \leq a < 3$ et l'algorithme se termine en 1 appel au plus. Pour la borne plus serrée du théorème, on considère le $b$ initiel : $F(n_2) \leq b < F(n_2 + 1)$ (affectation $a \leftarrow b$ lors du premier appel), et l'algorithme finit en $n_2 - 1$ appels au plus. ■

REMARQUE. La borne de théorème **??** montre le pire cas : c'est avec $a = F(n+1)$, $b = F(n)$. Avec un tel choix, $a \bmod b = F(n-1)$, et l'algorithme exécute $n - 2$ appels pour arriver à $a = F(3) = 2, b = F(2) = 1$ et se terminer après un dernier appel de plus où $b$ devient 0 et on retourne la réponse $a = 1$.

## 1.6 Tours de Hanoï

W(fr)

Parfois, la récursion nous fournit des solutions très simples à des problèmes complexes.



Dans le jeu de Tours de Hanoï, il faut déplacer des disques à diamètres différents $(1, 2, \ldots, n)$ d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire, tout en respectant Règles 1 et 2 ci-dessous. Opération MOVE$(i \to j)$ déplace le disque en haut de tour $i$ à tour $j$. Les disques sont en ordre décroissant au début, et il y a trois tours.

**Règle 1.** On ne peut déplacer plus d'un disque à la fois. Un déplacement consiste de mettre le disque supérieur sur une tour au-dessus des autres disques (s'il y en a).

**Règle 2.** On ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Une solution simple est fournie en définissant une procédure récursive HANOI$(i \curvearrowright k \curvearrowright j, n)$ qui déplace les $n$ disques supérieurs sur tour $i$ vers tour $j$ en utilisant la tour intermédiare $k$.

```
   HANOI(i ↷ j ↷ k, n)
H1 if n ≠ 0 then
H2     HANOI(i ↷ j ↷ k, n − 1)
H3     MOVE(i → j)
H4     HANOI(k ↷ i ↷ j, n − 1)
```

**Théorème 1.3.** *La procédure* HANOI *performe $2^n - 1$ déplacements pour arranger $n$ disques.*

*Démonstration.* La preuve est par induction en $n$. Soit $D(n)$ le nombre de déplacements (Ligne H3).

**Cas de base :** on vérifie que le théorème est valide pour $n = 0$ car $D(0) = 0 = 2^0 - 1$ et l'arrangement final est correct.

**Hypothèse d'induction :** supposons que le théorème est vrai pour un $n \geq 0$ quelconque.

**Cas inductif :** par inspection de l'algorithme, on a $D(n+1) = 2D(n) + 1$. En se servant de l'hypothèse d'induction, on a $D(n+1) = 2 \cdot \left(2^n - 1\right) - 1 = 2^{n+1} - 1$. On voit aussi que l'arrangement final est correct pour les $(n+1)$ disques. Donc le théorème est correct pour $n + 1$.

En conséquence, le théorème est correct pour tout $n = 0, 1, 2, \ldots$ . ■

**Exercice 1.3.** *Démontrer par induction que disque $m$ est déplacé $2^{n-m}$ fois par l'algorithme. (Notez que le résultat aussi montre que $D(n) = 2^0 + 2^1 + \cdots + 2^{n-1} = 2^n - 1$.)*

**Exercice 1.4.** *Démontrer par induction que $\sum_{k=0}^{n} F(k) = F(n+2)-1$ et que $\sum_{k=0}^{n} kF(k) = nF(n+2)-F(n+3)+2$.*
***Indice :*** *Démontrez les égalités pour les cas de base (il y en a deux !) et procédez dans le cas inductif en utilisant la définition* (1.2).