

# 1 Réursion

## 1.1 Factorielle

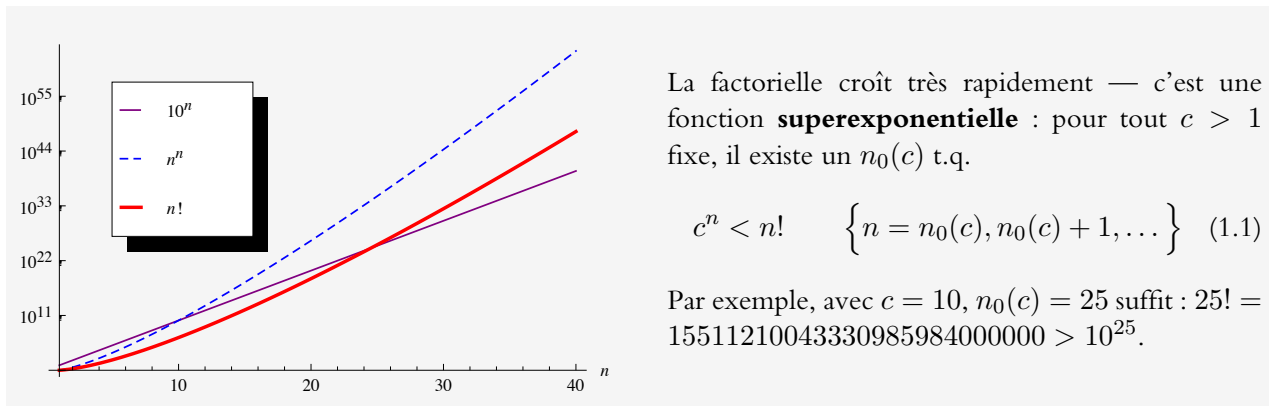
Définition de la factorielle :  $0! = 1$  et  $n! = 1 \times 2 \times 3 \times \dots \times n = \prod_{k=1}^n k$ . pour  $n > 0$ .

W<sup>(fr)</sup>

**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

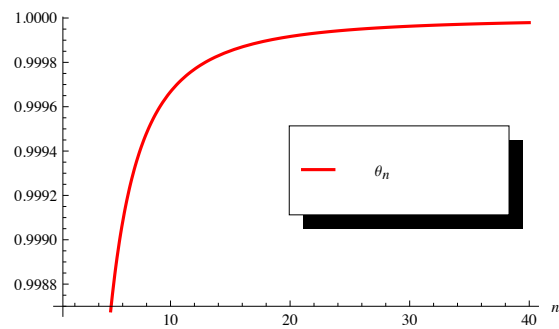
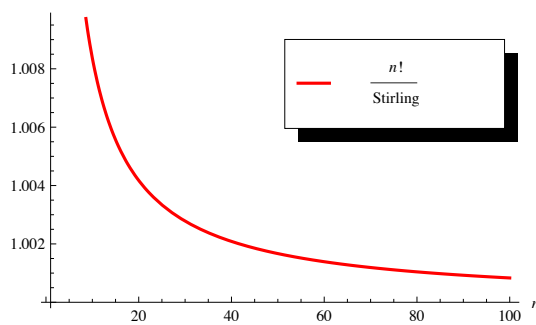


W<sup>(fr)</sup>

**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  $n_0(c) = \lceil ce \rceil$ ) :



### 1.3 Pile d'exécution

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

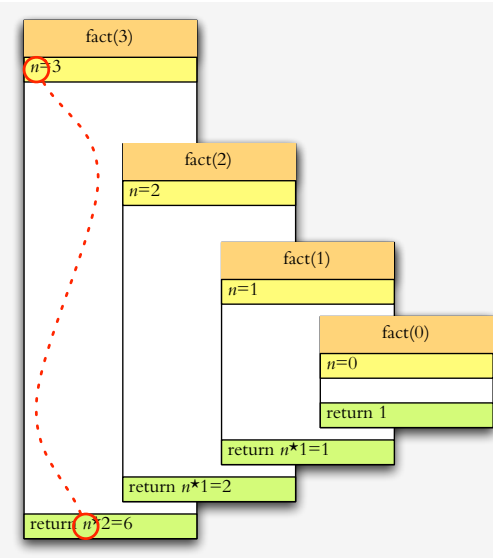
```

```

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

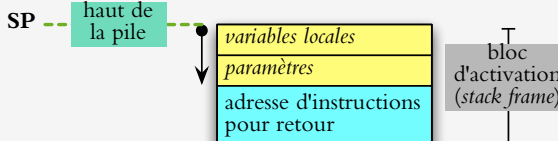
```

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 récupérer 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 jumponzero 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 relatif au bloc d'activation  $\Rightarrow$  chaque copie activée de la fonction possède ses propres variables locales.

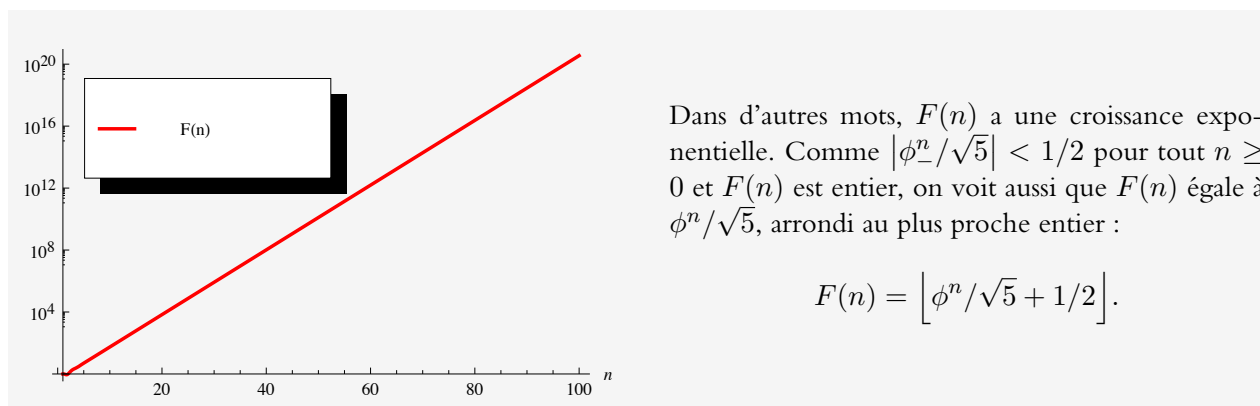
## 1.4 Nombres Fibonacci

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

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

**Formule de Binet.** Les racines de l'équation de récurrence homogène  $[x^n = x^{n-1} + x^{n-2}]$  sont  $\phi = \frac{1+\sqrt{5}}{2} = 1.618\dots$  et  $\phi_- = 1 - \phi = \frac{1-\sqrt{5}}{2} = -0.618\dots$ . La solution spécifique se trouve par la solution des équations  $F(0) = 0 = a\phi^0 + b\phi_-^0$  et  $F(1) = 1 = a\phi^1 + b\phi_-^1$ . On obtient  $a = -b = 5^{-1/2}$ , donc

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



## 1.5 Algorithme d'Euclide

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

```
GCD(a,b) // {b ≤ a}
E1 if b = 0 then return a
E2 else return GCD(b, a mod b);
```

```
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  $b$ .

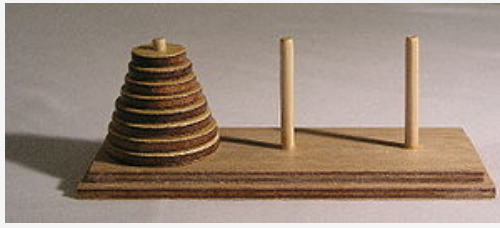
**Théorème 1.2.** Soit  $n$  le plus grand entier pour lequel  $F(n) \leq b < F(n+1)$  dans l'appel à l'algorithme d'Euclide. Alors, l'algorithme exécute au plus  $(n-1)$  récursions.

*Démonstration.* On définit  $n_i$  pour  $i = 1, 2, \dots$  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 initial). 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  $b$  initial :  $F(n_2) \leq b < F(n_2 + 1)$  (affectation  $a \leftarrow b$  lors du premier appel), et l'algorithme finit en  $n_2 - 1$  appels au plus. ■

REMARQUE. La borne de théorème 1.2 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

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  $(1, 2, \dots, n)$  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  $\text{MOVE}(i \rightarrow j)$  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.



W<sup>(fr)</sup>

**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 ou sur un emplacement vide.

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

```
HANOI( $i \curvearrowright j \curvearrowright k, n$ )
H1 if  $n \neq 0$  then
H2   HANOI( $i \curvearrowright j \curvearrowright k, n - 1$ )
H3   MOVE( $i \rightarrow j$ )
H4   HANOI( $k \curvearrowright i \curvearrowright j, n - 1$ )
```

**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  $D(n+1) = 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 1$ . 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  $n = 0, 1, 2, \dots$  ■