Cours 3 : Récursivité

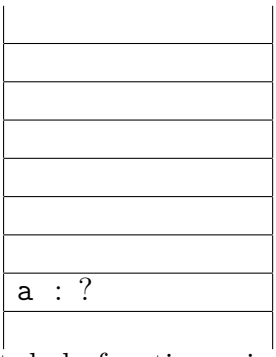
1. Définition et premiers exemples

1.1. **Rappel : la pile d'exécution.** Au cours de l'exécution d'un programme les variables sont stockées dans la mémoire sous forme d'une **pile**. A chaque appel de fonction, un espace mémoire est réservé pour les variables de la fonction.

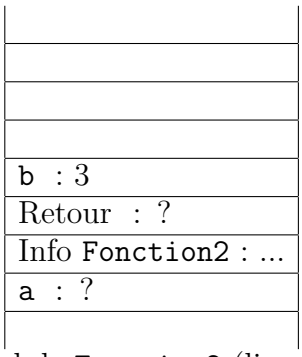
Exemple 1.1.

.

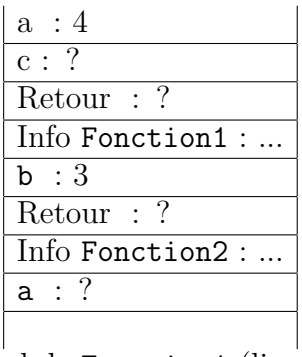
```
1  Fonction Fonction1(entier a):
2          Entier c <- a+1
3      Retourner c
4
5
6  Fonction Fonction2(entier b):
7      Retourner Fonction1(b+1)
8
9  Fonction Principale():
10         Entier a <- Fonction2(3)
11         ...
```



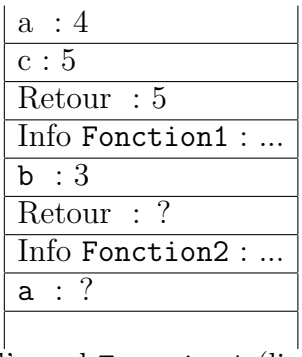
Début de la fonction principale



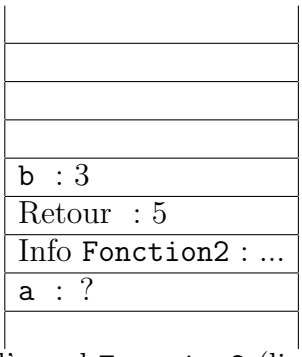
Appel de `Fonction2` (ligne 10)



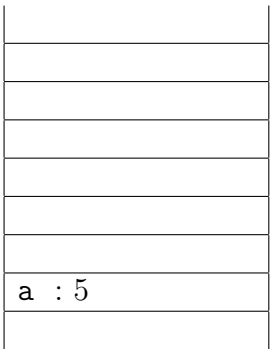
Appel de `Fonction1` (ligne 7)



Fin d'appel `Fonction1` (ligne 3)



Fin d'appel `Fonction2` (ligne 7)



Retour Fonction `Pincipale`

En particulier, rien n'empêche une fonction de s'appeler elle-même.

Exemple 1.2.

Que se passe-t-il au niveau de la pile dans l'exemple suivant ?

```
Fonction Fonction1(entier a):
    Fonction1(a)

Fonction Principale():
    Fonction1(3)
```

1.2. **Définition.**

Définition 1.3. *Un algorithme* récursif *est un algorithme qui fait appel à lui même. Lorsqu'un algorithme n'est pas récursif, on dit qu'il est* itératif.

Pour éviter une boucle infinie, il faudra impérativement définir **une condition d'arrêt**.

Exemple 1.4.
Le calcul de $n!$. Tout d'abord la version itérative

```
Fonction factorielle(entier n):
    Entier r <- 1
    Pour i allant de 1 à n:
        r <- r * i
    Retourner i
```

La version récursive se base sur le principe suivant :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}, \tag{1.1}$$

que l'on peut traduire sous forme d'algorithme.

```
Fonction factorielle(entier n):
    Si n = 0:
        Retourner 1
    Retourner n * factoriel(n-1)
```

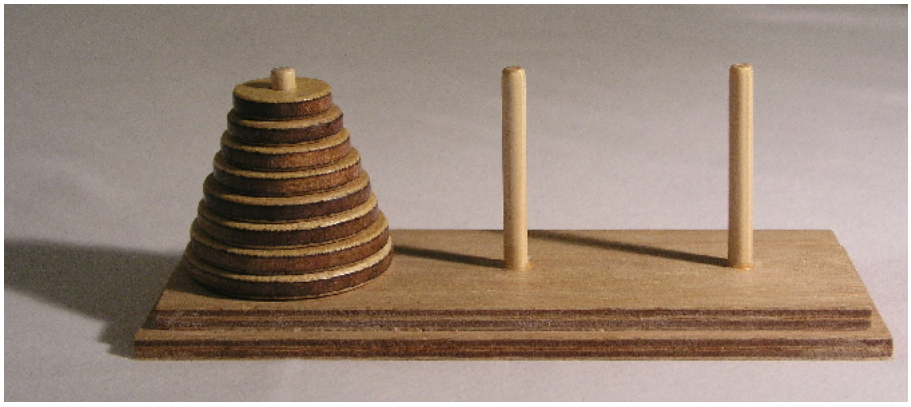
1.3. **Comment définir une fonction récursive.** Le principe d'une fonction récursive est assez similaire à la notion de **récurrence** mathématique. On ne cherche pas à calculer le problème dans son ensemble mais simplement à le définir en fonction d'un problème de taille plus petite. Tout comme en mathématique le fait de prouver l'état initial et une seule étape permet de prouver la propriété, dans un algorithme, le fait de calculer l'état initial et une étape permet de calculer n'importe quelle valeur.

Un algorithme récursif s'écrira toujours de cette façon :

```
Input : des paramètres
Processus:
    Si les paramètres correspondent à la condition d'arret (cas simple):
        Actions cas d'arret
    Sinon
        Appel(s) récursifs sur de nouveaux paramètres
```

Pour être sûr que l'algorithme termine, les nouveaux paramètres doivent se rapprocher de la condition d'arrêt.

Exemple 1.5.
Les tours de Hanoi



Problème : on possède trois pics sur lesquels sont empilés des disques de tailles décroissantes (tous sur le premier pic). On souhaite déplacer la pile sur le second pic en suivant les règles suivantes :

— on ne peut déplacer qu'un seul disque à la fois, celui du haut de la pile,
— on ne peut poser un disque que sur un disque plus grand.

On suppose donc que l'on dispose d'une seule action possible :

```
Fonction Déplacer
Input :
    - le pic de départ
    - le pic d'arrivée
Processus :
    Si départ n'est pas vide :
        D <- disque retiré au sommet de départ
    Sinon :
        Erreur "Départ est vide"
    Si D < disque au sommet d'arrivée :
        Poser D au sommet d'arrivée
    Sinon :
        Erreur "Action impossible"
```

Comment résoudre ce problème ?

Pour résoudre un problème de façon récursive, il faut répondre à plusieurs questions :

(1) Quelle est la taille de mon problème ?
(2) Quelle est la plus petite taille possible ? (condition d'arrêt)
(3) Comment résoudre le problème dans ce cas là ? (action d'arrêt)
(4) Si je suppose que je sais résoudre le problème pour toutes les tailles $k < n$, comment le résoudre pour la taille n ?

Dans le cas des tours de Hanoi, cela donne l'algorithme suivant :

```
Fonction Hanoi
Input :
    - n, le nombre de disque à déplacer
    - p1, le pic de départ
    - p2, le pic d'arrivée
    - p3, le pic que je peux utiliser comme intermédiaire
Processus :
    Si n = 0 :
        Stop
    Hanoi(n-1,p1,p3,p2)
    Déplacer(p1,p2)
    Hanoi(n-1,p3,p2,p1)
```

2. Complexité d'un algorithme récursif

2.1. **Méthode de calcul.** Reprenons le schéma de base d'un algorithme récursif que l'on précise légèrement :

```
Fonction R
Input : des paramètres
Processus :
    Si les paramètres correspondent à la condition d'arret (cas simple):
        Actions cas d'arret
    Sinon
        Actions itératives et
        k Appel(s) récursifs : R(p1), R(p2), ..., R(pk)
```

On veut compter le nombre d'actions de base. Chaque appel de fonction est une action de base. Soit f la complexité de l'algorithme. Pour simplifier l'écriture, on suppose que les paramètres de l'algorithme correspondent à la taille n du problème. La condition d'arrêt est obtenue quand $n = 0$ et correspond à une complexité constante c_0. On obtient une définition récursive de f

$$f(0) = c_0 \tag{2.1}$$

$$f(n) = g(n) + \sum_{i=0}^{k} f(p_i) \tag{2.2}$$

où g est la complexité de la partie itérative. La complexité dépend donc du **nombre d'appels récursifs** à chaque étape et de la **taille des paramètres**. Il peut être plus ou moins simple de *développer* la fonction f pour obtenir une formule close (non récursive).

2.2. **Cas classiques.**

Exemple 2.1.
Cas linéaire

```
Fonction R
Input : un entier n
Processus :
    Si n=0 :
        Actions cas d'arret
    Sinon
        Appel de R(n-1)
```

Dans ce cas on a

$$f(0) = O(1) \tag{2.3}$$

$$f(n) = 1 + f(n-1) \tag{2.4}$$

Lorsqu'on développe, on obtient

$$f(n) = 1 + 1 + 1 + \cdots + 1 \tag{2.5}$$

n fois.

Conclusion : la complexité est en $O(n)$.

Exemples : fonction factorielle.

Exemple 2.2.
Cas exponentiel

```
Fonction R
Input : un entier n
Processus :
    Si n=0 :
        Actions cas d'arret
    Sinon :
        2 appels de R(n-1)
```

Dans ce cas on a

$$f(0) = O(1) \tag{2.6}$$
$$f(n) = 1 + 2 * f(n-1). \tag{2.7}$$

Par exemple, pour $n = 3$:

$$f(3) = 1 + 2f(2) = 1 + 2(1 + 2f(1)) = 1 + 2(1 + 2(1 + 2)) \tag{2.8}$$
$$= 15 = 2^4 - 1. \tag{2.9}$$

On prouve facilement par récurrence que

$$f(n) = 2^{n+1} - 1. \tag{2.10}$$

On est dans le cas d'une complexité exponentielle en $O(2^n)$.
Exemples : Tour de Hanoi, fibonacci

Exemple 2.3.
Cas logarithmique

```
Fonction R
Input : un entier n
Processus :
    Si n=0 :
        Actions cas d'arret
    Sinon :
        Appel de R(n/2)
```

Dans ce cas on a

$$f(0) = O(1) \tag{2.11}$$
$$f(n) = 1 + f(n/2). \tag{2.12}$$

Développons sur un exemple :

$$f(10) = 1 + f(5) = 2 + f(2) = 3 + f(1) = 5 \tag{2.13}$$
$$= log(10) + 2. \tag{2.14}$$

On prouve par récurrence

$$f(n) = \lfloor log(n) \rfloor + 2. \tag{2.15}$$

On a donc une complexité en $O(log(n))$.
Exemples : Recherche dichotomique, puissance (deuxième version du TD)

2.3. **Itératif ou récursif ?** En terme de complexité, la récursivité ne permet pas par défaut d'obtenir des algorithmes plus efficaces. Quand elle est mal utilisée, elle peut même donner des complexité **plus mauvaise** : exemple de Fibonacci. Par ailleurs, on peut prouver qu'il existe **toujours** une version itérative d'un algorithme récursif : il suffit de déplier la pile !

Qu'en est-il de la complexité mémoire ? Reprendre l'exemple de la fonction exponentielle en itératif et en récursif. La complexité mémoire de l'algorithme récursif est en $O(n)$ tandis qu'il est en $O(1)$ pour l'itératif.

Alors pourquoi on fait du récursif ? Sur de nombreux problèmes, les algorithmes récursifs sont beaucoup plus simples à concevoir que les algorithmes itératifs. C'est le cas de l'algorithme des Tour de Hanoi. Ils donnent des codes plus courts et plus lisibles. Par ailleurs, certaines **structure de données** ont elles-mêmes des **définition récursives** et sont particulièrement adaptés aux algorithmes récursifs : les arbres, les graphes, les listes chaînées.