Chapitre 9

Récursivité

Chapitre 9 La récursivité

I Fonctions récursives

Utilisation : Elles permettent de réaliser les suites récurrentes.

I.1) Un exemple basique : La fonction factorielle

N! = 1*2*3*4*....*(N-1)*N ou N! = N*(N-1)*(N-2)*....*2*1

Soit l'algo itératif qui réalise cette fonction :

fonction Facto (n : entier) : entier	**Ord**	**Lexique**
F ← 1		i (entier) : compteur
pour i ← 1 **à** n **faire**		
| F ← F * i		F (entier) : résultat
fpour		
Facto ⇐ F		

Chapitre 9 La récursivité

Rappel : Facto(0)= Facto(1) = 1

Facto(n)= Facto(n-1)*n, avec n ≥ 0

Décomposition : F ← F*i

On peut calculer F(n) si on connaît F(n-1)

On peut calculer F(n-1) si on connaît F(n-2)

……..

On peut calculer F(1) si on connaît F(0) = 1

Le calcul s'arrête lorsque la valeur de n = 0 ou n = 1

Chapitre 9 La récursivité

Nouvelle formulation :

fonction Facto (n : entier) : entier	**Ord**	**Lexique**
si n ≤ 1 **alors**		
| Facto ⇐ 1		
sinon		
| Facto ⇐ Facto(n-1)*n		
Fsi		

- On appelle la même fonction sur une valeur différente.
- Elle fait appel, ou référence, à elle-même pour se définir.
- Une telle fonction est dite fonction **récursive**.

Chapitre 9 La récursivité

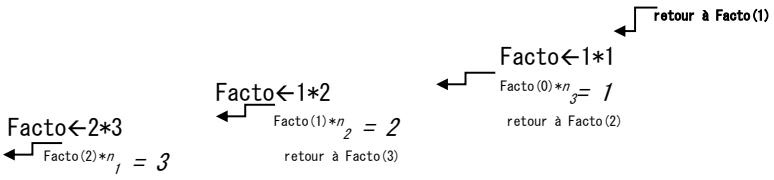
Exécution :

Prenons l'exemple du calcul de Facto(3) : $F_n \leftarrow F_{n-1} * n$

(hypothès

	1ère activation	2ème activation	3ème activation	4ème activation
Facto(3) ⇒	$n_1 = 3$	$n_2 = 2$	$n_3 = 1$	$n_4 = 0$
	Facto(2) ⇒	Facto(1) ⇒	Facto(0) ⇒	1

= 6



Chapitre 9 La récursivité

Exemple d'utilisation :

algo Calcul_Facto	**Ord**	**Lexique**
ecrire(Facto(3))		Facto(**fonction**/entier) : factorielle

Les variables n_1, n_2, n_3, n_4 sont créées lors des activations successives de la fonction **Facto**. Elles ne sont utilisées qu'au retour du niveau d'activation suivant, *c.a.d* dans l'ordre **inverse** où elles ont été créées.

Ce sont des variables mémoires gérées en pile (*First In, Last Out* **ou** *Last In, First Out*) : *premier créé dernier utilisé (ou dernier ….)*

La pile d'exécution est un emplacement mémoire destiné à conserver les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution.

Chapitre 9 La récursivité

Dans un algorithme :

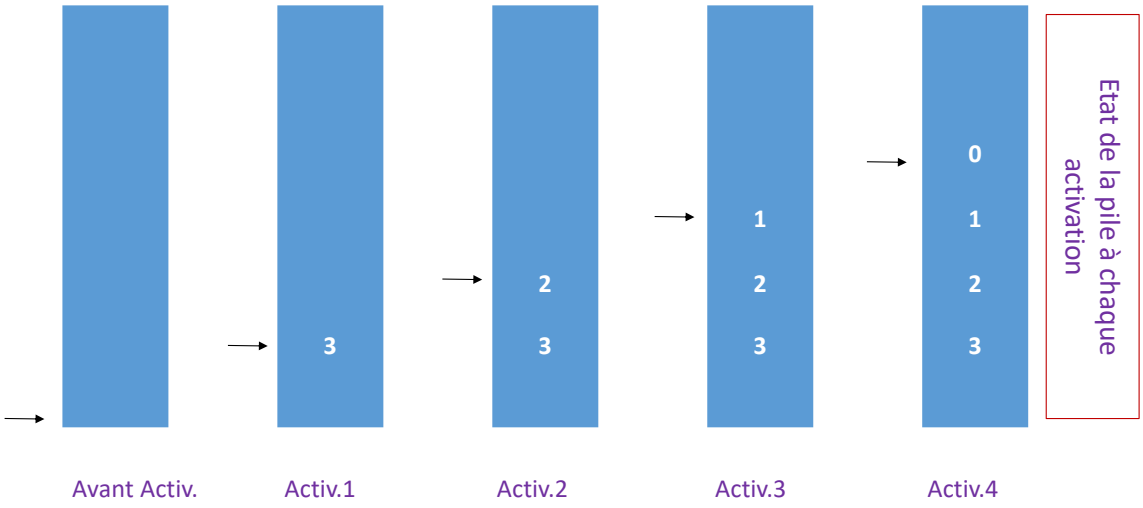
A chaque appel de fonction (ou module) ⇒ création d'une pile d'exécution propre à l'appel en cours.

- Au départ ⇒ elle est vide
- A chaque appel ⇒ tous les objets qui composent l'environnement d'exécution sont empilés
- les appels s'arrêtent lorsque la pile est vide

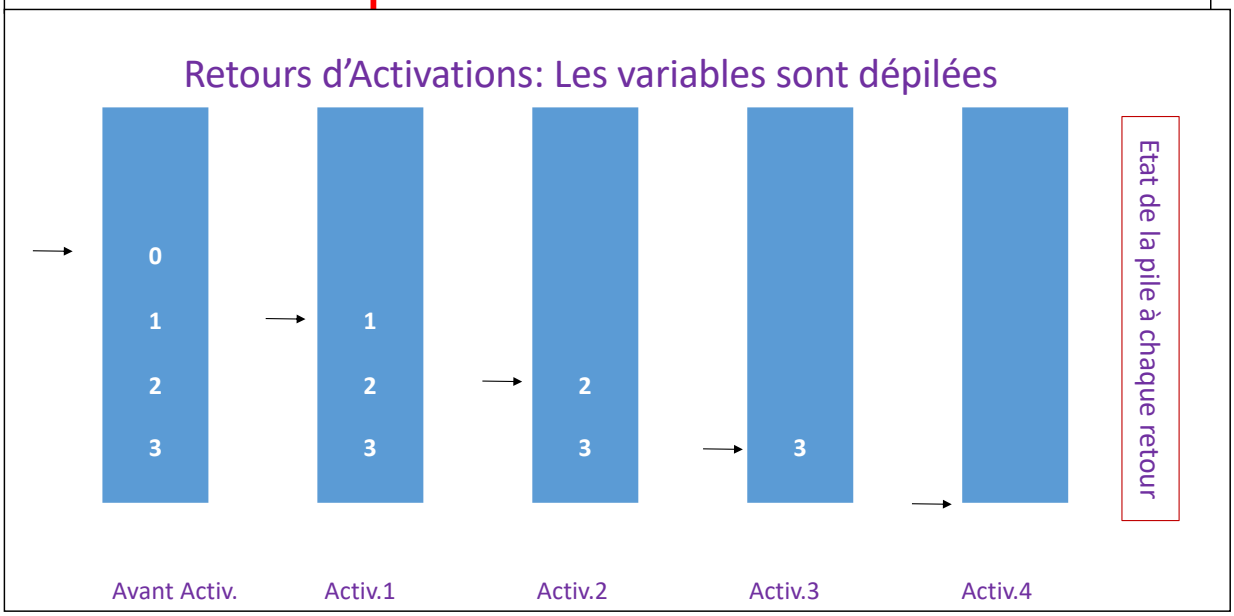
(*ou encore : Lorsque les appels sont terminés ⇒ La pile est vide*)

Chapitre 9 La récursivité

Activations successives : Les variables sont empilées



Chapitre 9 La récursivité



Chapitre 9 La récursivité

I.2 Définition

- Une fonction récursive **FR** est une fonction dont le résultat est défini à partir d'un ou plusieurs appels d'elle-même dans l'algorithme qui la définit. Les appels *récursifs* s'arrêtent lorsqu'une certaine condition, dite condition d'arrêt, est vérifiée.
- On peut définir sa forme comme suit :

FR = Fonction(X, **FR**)

où X est un ensemble de ***règles*** ou de ***fonctions*** connues et Fonction une combinaison de X et de **FR**.

- On parle de *fonction récursive* ***terminale*** si tout appel récursif à la fonction ne nécessite aucun calcul ni composition de calculs sur la fonction.

Exemple :

La fonction factorielle n'est pas une fonction récursive terminale car sa dernière instruction n'est pas un appel pur de la fonction mais une combinaison d'un appel et d'un produit.

Chapitre 9 La récursivité

I.3 Performances

- La grande majorité des langages de programmation intègrent cette technique de programmation, correspondant à ce type de raisonnement (**récursion**) qui est extrêmement puissante.
- Elle est *élégante* dans l'écriture car elle permet d'aboutir à des programmes concis et proches de la formulation mathématique du problème.
- Mais c'est une forme de programmation généralement *gourmande* en place mémoire et en temps d'exécution :
  - ✓ chaque appel récursif oblige le processeur à différer les traitements et calculs en cours en empilant des variables *locales* à l'appel courant.
  - ✓ La pile peut être rapidement saturée en fonction du nombre d'appels (correspondant à la profondeur de la *récursion*).
- On l'utilise quand on ne peut résoudre le problème de manière itérative.
- On l'utilise quand on ne peut faire autrement surtout s'il s'agit de résoudre des problèmes portant sur des données de type récursif. ***Exemple*** : le type arbre, le type liste,…

Chapitre 9 La récursivité

Remarque :

Il est prouvé que tout algorithme *récursif* peut être remplacé par un algorithme *itératif* (de type récurrent : usage de boucles ou itérations conditionnelles ou non) essentiellement dans le cas de récursion terminale.

Chapitre 9 La récursivité

I.V Exemple

Soit la chaîne "12345".

On veut déterminer la chaîne ***miroir*** (inverse) de cette chaîne : "12345" ⇒ "54321"

Soit la fonction récursive Miroir qui trouve le miroir de cette chaîne.

Miroir ("12345") = Miroir("2345") | '1'

Miroir ("2345") = Miroir("345") | '2'

Miroir ("345") = Miroir("45") | '3'

Miroir ("45") = Miroir("5") | '4'

Miroir ("5") = '5'

ou

Miroir("5") = Miroir("") | '5' **et** Miroir("") = ""

Chapitre 9 La récursivité

fonction Miroir (ch : chaine) : chaine	**Ord**	**Lexique**
si longueur(ch) > 1 **alors** | Miroir ⇐ Miroir(ch[2..**longueur**(ch)])|ch[1] **sinon** | Miroir ⇐ ch **fsi**		

NB : *concaténation* ⇒ |

Chapitre 9 La récursivité

II Module récursif

II.1 Définition

Un module récursif est un module qui est défini à partir d'un ou de plusieurs appels de lui-même.

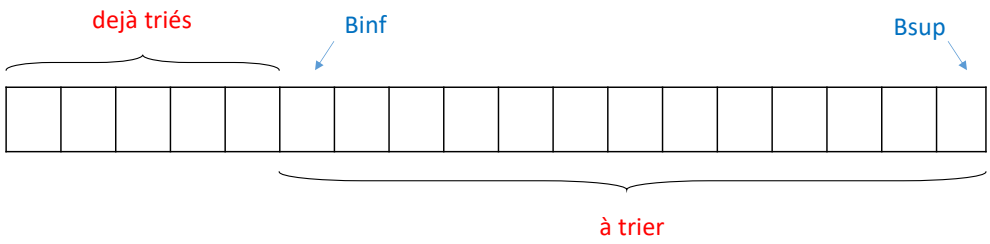
II.2 Exemples

a) Le tri

On utilisera le tri par échange.

L'exemple du tri d'une table illustre un autre principe de récurrence lié à l'utilisation d'un type de données à forme récurrente : la table.

Chapitre 9 La récursivité



Principe **:**

- La table est triée jusque l'élément n° (*binf -1*) où *binf* est l'indice de la première case non triée
- On cherche le minimum entre *binf* et *bsup*
- On l'échange avec la case n° *binf*
- On recommence selon le même principe à partir de *binf*+1
- On s'arrête quand le nombre d'éléments à trier = 0 (ou =1)
  (une table où il y a un seul élément à trier est déjà triée)

Chapitre 9 La récursivité

module TriEchange (↓ binf,bsup:entier; ↕ T:T_Tabent)	**Ord**	**Lexique**
si bsup ≠ binf **alors** | Echange (T[binf], T[RechPosmin(binf, bsup,T)]) | TriEchange(binf+1,bsup,T) | **fsi**		Echange (module) : *échange le contenu de deux cases de la table* RechPosmin(fonction /entier) : *calcule la position du minimum*

Chapitre 9 La récursivité

Exemple :

Soit le tableau T à trier suivant :

1	2	3	4
3	5	1	2

- TriEchange(1, 4, (3-5-1-2)) ⇒
- ✓ Echange(T[1],T[3])
  - TriEchange(2, 4, (1-5-3-2)) ⇒
- ✓ Echange(T[2], T[4])
  - TriEchange(3, 4, (1-2-3-5)) ⇒
- ✓ Echange(T[3], T[3])
  - TriEchange(4, 4, (1-2-3-5)) ⇒ fin car la condition d'arrêt est vérifiée

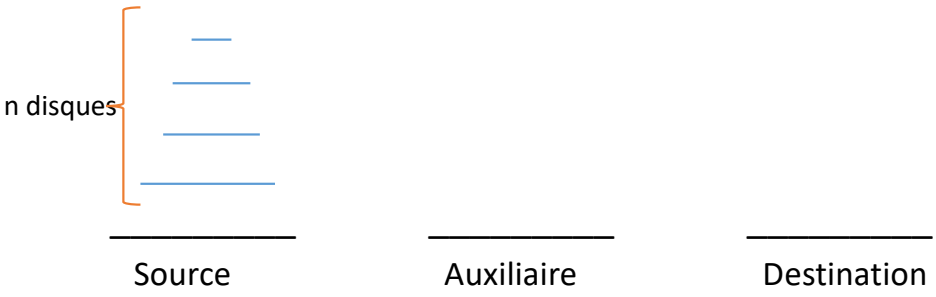
Chapitre 9 La récursivité

b) Les tours de hanoï

Chapitre 9 La récursivité

- On dispose de 3 tours et de ***n*** disques de tailles différentes
- Les ***n*** disques sont disposés sur une tour
- On ne peut déplacer qu'un seul disque à la fois
- On ne peut empiler un disque sur un autre de taille plus petite
- On veut déplacer les disques d'une tour à une autre

Chapitre 9 La récursivité

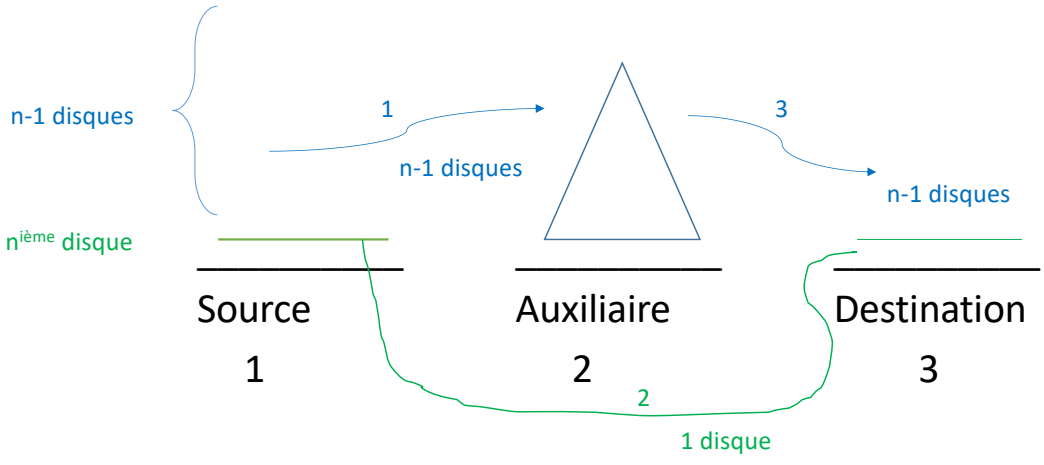


Hypothèses :

- On sait déplacer (*n-1*) disques d'une tour (*source*) à une autre (*destination*)
- On sait faire le déplacement d'un disque

Chapitre 9 La récursivité

3 déplacements :



Chapitre 9 La récursivité

Il en ressort les 3 étapes suivantes :

1. Déplacer n-1 disques de Source ⇒ Auxilliaire
2. Déplacer 1 disque vers Destination : action élémentaire
3. Déplacer n-1 disques de Auxiliaire ⇒ Destination

Chapitre 9 La récursivité

module Hanoi (↓n : entier; ↓Source, Aux, Dest : entier)	**Ord**	**Lexique**
si n ≠ 0 **alors**		
|		
| Hanoi(n-1, Source, Dest, Aux)		
|		
| Deplacer (1, Source, Dest)		
|		
| Hanoi (n-1, Aux, Source, Dest)		
fsi		

Remarque :

- Pour n disques, on effectue 2^n-1 déplacements.
- Ce problème a été posé par le mathématicien Lucas Edouard [1842-1891])