CHAPITRE 2

LA RÉCURSIVITÉ

Comment une fonction peut se rappeler elle-même

LA RÉCURSIVITÉ

- Introduction
- Exemple
- Contre-exemple
- La méthode itérative

INTRODUCTION

- **<u>Définition :</u>** Un traitement (programme, fonction, procédure ou structure de données) récursif est un traitement qui s'appelle lui-même, à l'opposé d'un traitement itératif qui ne le fait pas.
- Cette notion est présente partout en informatique. Le 1er exemple en est la structure des fichier sur une partition, dans laquelle un répertoire, un dossier, est composé de fichiers dont certains sont eux-mêmes des répertoires.
- En programmation, elle permet de simplifier l'écriture de certains traitements.
- Par définition, afin d'assurer que la récursion se termine, ***un traitement, une fonction, devra se rappeler avec un paramètre variable qui devra atteindre une condition d'arrêt***.
  Par exemple, pour un répertoire, ce sera le chemin d'accès à ce répertoire qui permettra de se repérer dans l'arborescence des dossiers.

EXEMPLE (1/2)

- **Exemple :** Calcul de la factorielle d'un nombre entier n.

```
Fonction FactorielleRécursive  (N : entier) : entier
Début
     Si N <= 1 Alors
          Retourner  1
     Sinon
          Retourner  (N * FactorielleRécursive (N – 1))
     Fin Si
Fin FactorielleRécursive
```

EXEMPLE (2/2)

- Pile des valeurs d'appel (→) et de retour (←) :

A	3→		
B		2→	
C			1→
D			1←
E		2←	
F	6←		

- Empilement des appels :
(***pile d'exécution***)

A	B	C	D	E	F
N=3	N=2 N=3	N=1 N=2 N=3	N=2 N=3	N=3	
					N=6

- Le principe du traitement précédemment décrit correspond à la gestion d'une **PILE** :
  - ✓ Les traitements à exécuter sont empilés les uns au dessus des autres.
  - ✓ Ils sont exécutés en retour dans l'ordre inverse de leur ordre d'arrivée.
  - ✓ C'est le principe du **LIFO** (Last In First Out).
- REMARQUES : La méthode **FIFO** (First In First Out) consiste au contraire à traiter en premier la première entité stockée.

LA MÉTHODE ITÉRATIVE

```
Fonction FactorielleIterative  (N : entier) : entier
Début
     i ← N
     Tant Que  i > 1  Faire
          i ← i – 1
          N ← N * i        // au 1er tour ⇔ N * (N-1)
     Fin Tant Que
     Retourner N
Fin FactorielleItérative
```

- La procédure Factorielle est appelée :
  - ✓ Le contenu de la variable N (factorielle à calculer) est stocké dans la variable i.
  - ✓ Tant que le contenu de la variable i est supérieur à 1 :
    - i est décrémenté de 1.
    - Les contenus de i et N sont multipliés et le résultat est stocké dans N.
  - ✓ On retourne le contenu de N à la fin

CONTRE-EXEMPLE (1/3)

- **Exemple :** Calcul de la suite de Fibonacci : F(0) = 1
  F(1) = 1
  F(N) = F(N-1) + F(N-2)

```
Fonction FiboRécursive  (N : entier) : entier
Début
     Si N <= 1 Alors
          Retourner  1
     Sinon
          Retourner  FiboRécursive (N-1) + FiboRécursive(N-2)
     Fin Si
Fin FactorielleRécursive
```

CONTRE-EXEMPLE (2/3)

- **Exemple :** Calcul de la suite de Fibonacci : F(0) = 1
  F(1) = 1
  F(N) = F(N-1) + F(N-2)

- Les appels identiques vont se multiplier et se recalculer de nombreuses fois.
  Exemple :

```
FR(N) = FR(N-1) + FR(N-2)
FR(N) = (FR(N-2) + FR(N-3)) + FR(N-2)
FR(N) = 2*FR(N-2) + FR(N-3)
FR(N) = 2*(FR(N-3) + FR(N-4)) + FR(N-3)
FR(N) = 3*(FR(N-3) + 2*FR(N-4)
FR(N) = 3*(FR(N-4) + FR(N-5)) + FR(N-4)
FR(N) = 4* FR(N-4) + 3*FR(N-5)
……
```

- Si on continue, on va appeler de nombreuses fois le même calcul.
- C'est pourquoi des algorithmes complémentaires retiennent ces similarités pour ne pas les reproduire

CONTRE-EXEMPLE (3/3)

```
Fonction FibonacciIterative  (N : entier) : entier
Variables a, b, c : entier
Début
     a ← 1
     b ← 1
     Pour i de  2 à N  Faire
          c ← a + b
          b ← a
          a ← c
     Fin Tant Que
     Retourner a
Fin FibonacciIterative
```

- La procédure **`FibonacciIterative`** est appelée :
  - ✓ On part de la base avec F(1)=F(2)=0
  - ✓ A chaque tour de boucle, on calcule une valeur de plus pour arriver à F(N)
  - ✓ Le calcul est linéaire et ne nécessite pas de rappeler plusieurs fois la fonction avec un paramètre identique.
  - ✓ On a ainsi, dans ce cas, une complexité linéaire et non exponentielle de calcul en espace et temps. Mais c'est un exemple simple qui n'est pas toujours réalisable dans la pratique.

CONCLUSION

- La récursivité est une des notions les plus puissantes de l'informatique. Elle permet de programmer de manière quasi identique à la description de la fonction voulue.
- On oppose la méthode "récursive" à la méthode "itérative".
- La méthode récursive laisse le programme gérer les différents paramètres d'appels de fonction, par conséquent plus génériques et donc plus lourds. De plus, l'empilement successif des appels peut amener des problèmes de mémoire ou d'efficacité, donc de rapidité, voir d'impossibilité de solution par dépassement de temps ou mémoire.
- La méthode itérative, elle, ne fait pas appel à la récursivité pour obtenir le résultat. De ce fait, elle est toujours plus efficace car nécessitant moins de ressources (en temps et en espace), mais elle est souvent plus (voir très) difficile à programmer car elle demande, la plupart du temps, de gérer soi-même la pile d'exécution et les variables de gestion.
- Au final, la méthode récursive est plus simple à programmer et, dans certains cas, elle sera la seule solution possible.
- A l'inverse, lorsque l'espace demandé est trop important ou la récursion inefficace, seule la solution itérative pourra être utilisée.