Chapitre 2

Récursivité

2.1 Principe

En mathématiques, les définitions par récurrence sont assez courantes. Par exemple la suite $F(n)$ suivante dite de Fibonacci est récurrente car la fonction elle même intervient dans sa définition :

$$\begin{cases} F(0) = F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \quad \forall n > 1 \end{cases}$$

En informatique, le récursivité est un concept important, elle représente un style élégant de programmation. Elle consiste à subdiviser un problème en plusieurs sous-problèmes puis résoudre chaque sous-problème et combiner ensuite les petites solutions pour construire la solution globale. Ce principe est appelé "*Diviser pour régner*" ou "*Diviser pour résoudre*".

2.2 Algorithme récursif

Un programme (une fonction, une procédure) est dit récursif s'il s'appelle lui même, c'est à dire s'il contient un appel à lui même dans sa définition, sinon il est appelé itératif.

Dans l'exemple suivant, on désire calculer la somme des entiers de 0 jusqu'à n. La fonction $SommeI$ calcule la somme d'une façon itérative alors que la fonction $SommeR$ la calcule d'une façon récursive selon le principe suivant : "*La somme des entiers de 0 jusqu'à n est égal à n plus la somme des entiers de 0 à* $n-1$".

$$\sum_{i=0}^{n} = n + \sum_{i=0}^{n-1}$$
$$\sum_{i=0}^{0} = 0$$

```
Fonction SommeI( n : entier) : entier;
Var  S,i : entier ;
Début
    S ← 0 ;
    Pour i de 1 à n faire
        S ← S + i ;
    Fin Pour;
    SommeI ← S ;
Fin;
// Fonction somme itérative
```

```
Fonction SommeR( n : entier) : entier;
Début
    Si (n = 0) Alors
        SommeR ← 0 ;
    Sinon
        SommeR ← n + SommeR(n−
        1) ;
    Fin Si;
Fin;
// Fonction somme récursive
```

L'exécution d'un appel $SommeR(5)$ par exemple se fait comme suit :

- Appel à $SommeR(5)$
  - 5 + ($SommeR(4) = ?$)
  - Appel à $SommeR(4)$
    - 4 + ($SommeR(3) = ?$)
    - Appel à $SommeR(3)$
      - 3 + ($SommeR(2) = ?$)
      - Appel à $SommeR(2)$
        - 2 + ($SommeR(1) = ?$)
        - Appel à $SommeR(1)$
          - 1 + ($SommeR(0) = ?$)
          - Appel à $SommeR(0)$
            - **Retour de 0**
        - Retour de 1 (1 + 0)
      - Retour de 3 (2 + 1)
    - Retour de 6 (3 + 3)
  - Retour de 10 (4 + 6)
- Retour de 15 (5+10)

La sauvegarde des différentes valeurs lors de ces appels se fait dans la pile d'exécution du programme.

> La **pile d'exécution** du programme en cours est un emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution. Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

La pile du programme PP appelant la fonction $SommeR(3)$ contiendra l'adresse de l'instruction d'appel soit @RetPP et les valeurs de ses variable (VV) à l'instant de l'appel. De même un appel à partir de la fonction $SommeR$ va ajouter à la pile l'adresse de la fonction $SommeR$, l'adresse de l'instruction d'appel, soit @RetR et la valeur de n. L'évolution de la pile sera donc somme suit :

					SommeR, @RetR, n=0						
				SommeR, @RetR, n=1	SommeR, @RetR, n=1	SommeR, @RetR, n=1					
			SommeR, @RetR, n=2	SommeR, @RetR, n=2	SommeR, @RetR, n=2	SommeR, @RetR, n=2	SommeR, @RetR, n=2				
		SommeR, @RetR, n=3	SommeR, @RetR, n=3	SommeR, @RetR, n=3	SommeR, @RetR, n=3	SommeR, @RetR, n=3	SommeR, @RetR, n=3	SommeR, @RetR, n=3	SommeR, @RetR, n=3		
	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	PP, @Ret, VV	
t=0	1	2	3	4	5	6	7	8	9	10	11

Attention ! La pile a une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile (stack overflow).

2.3 Terminaison

Comme dans le cas d'une boucle, il faut un cas d'arrêt où l'on ne fait pas d'appel récursif. L'évolution des appels récursifs doit impérativement aboutir à un cas initial où on ne fait pas d'appels récursifs. Dans le cas contraire les appels seront infinis et il y aura un débordement de la pile (stack overflow). La fonction suivante, par exemple, produit un débordement de la pile avec l'appel $SommeR(2)$ puisque n commence de 2 et est incrémenté

à chaque appel et la condition $(n = 0)$ ne sera jamais satisfaite :

```
Fonction SommeR( n : entier) : entier;
Début
    Si (n = 0) Alors
        SommeR ← 0 ;
    Sinon
        SommeR ← n + SommeR(n + 1) ;
    Fin Si;
Fin;
```

2.4 Exemples

2.4.1 La factorielle

Factorielle n, noté $n!$, est le produit de tous les entiers de 1 jusqu'à n, soit $1 \times 2 \times 3 \times 4 \times ... \times n$

Par exemple : 5 ! = 1 * 2 * 3 * 4 * 5 = 120

De plus : $n! = (n-1)! \times n$ et

Avec le même exemple : 5 ! = 4 !*5

```
Fonction FactI( n : entier) : entier;
Var  M,i : entier ;
Début
    M ← 1 ;
    Pour i de 1 à n faire
        M ← M × i ;
    Fin Pour;
    FactI ← M ;
Fin;
// Fonction factorielle itérative
```

```
Fonction FactR( n : entier) : entier;
Début
    Si (n ≤ 1) Alors
        FactR ← 1 ;
    Sinon
        FactR ← n × FactR(n − 1) ;
    Fin Si;
Fin;
// Fonction factorielle récursive
```

2.4.2 Le PGCD

Le Plus Grand Commun Diviseur (PGCD) de deux entiers A et B est calculé en faisant des soustractions successives entre A et B jusqu'à arriver à deux entier égaux qui représentent le PGCD.

Par exemple :

A	42	18	18	12	6
B	24	24	6	6	6

On remarque que le PGCD de 42 et 18 est le même que celui de 18 et 24 et le même que celui de 18 et 6 ...et ainsi de suite, d'où la définition récursive :

$$PGCD(A,B) = \begin{cases} A & si\ A = B \\ PGCD(A-B,B) & si\ A > B \\ PGCD(A,B-A) & sinon \end{cases}$$

Et d'où la fonction récursive $PGCDR$:

```
Fonction PGCDI( A,B : entier) : entier;
Début
    Tant que (A ≠ B) faire
        Si (A > B) Alors
            A ← A − B
        Sinon
            B ← B − A
        Fin Si;
    Fin TQ;
    PGCDI ← A ;
Fin;

// Fonction PGCD itérative
```

```
Fonction PGCDR( A,B : entier) : entier;
Début
    Si (A = B) Alors
        PGCDR ← A ;
    Sinon
        Si (A > B) Alors
            PGCDR ← PGCDR(A−B,B) ;
        Sinon
            PGCDR ← PGCDR(A,B−A) ;
        Fin Si;
    Fin Si;
Fin;
// Fonction PGCD récursive
```

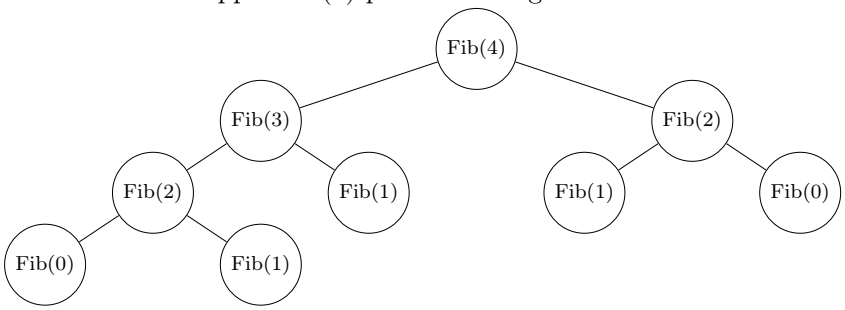
2.4.3 Suite de Fibonacci

Le calcul de la suite de Fibonacci vue précédemment est récursif de nature. L'écriture de l'algorithme correspondant est intuitive :

```
Fonction Fib( n : entier) : entier;
Début
    Si (n ≤ 1) Alors
        Fib ← 1 ;
    Sinon
        Fib ← Fib(n − 1) + Fib(n − 2) ;
    Fin Si;
Fin;

// Fonction Fibonacci récursive
```

L'exécution de l'appel $Fib(4)$ peut être imaginé comme suit :



Remarque : La solution récursive n'est pas toujours la plus efficace. Par exemple, dans le cas de cette fonction le calcul de Fib(1) se fait plusieurs fois.

2.4.4 Le tri par fusion

Le tri par fusion permet de trier rapidement un tableau. L'algorithme est naturellement décrit de façon récursive.

- Si le tableau n'a qu'un seul élément, il est déjà trié.
- Sinon, séparer le tableau en deux parties à peu près égales.
- Trier récursivement les deux parties avec l'algorithme du tri fusion.
- Fusionner les deux tableaux triés en un seul tableau trié.

L'algorithme est le suivant :

```
Type Tab = Tableau[1..N] de entier ;
Procédure Fusion(var T : Tab ; Debut, Milieu, Fin : entier);
var T1,T2 : Tab ;
    i,j,k,n1,n2 : entier ;
Début
    n1 ← Milieu - Debut +1 ;
    n2 ← Fin - Milieu ;
    Pour i de 1 à n1 faire
        T1[i] ← T[i + Debut -1] ;
    Fin Pour;
    Pour j de 1 à n2 faire
        T2[j] ← T[j + Milieu] ;
    Fin Pour;
    i ← 1 ; j ← 1 ;
    Pour k de Debut à Fin faire
        Si (i>n1) Alors
            T[k] ← T2[j] ;
            j ← j+1 ;
        Sinon
            Si (j>n2) Alors
                T[k] ← T1[i] ;
                i ← i+1 ;
            Sinon
                Si (T1[i]<T2[j]) Alors
                    T[k] ← T1[i] ;
                    i ← i + 1 ;
                Sinon
                    T[k] ← T2[j] ;
                    j ← j+1 ;
                Fin Si;
            Fin Si;
        Fin Si;
    Fin Pour;
Fin;
```

```
Procédure Tri(var T : Tab ; Debut, Fin : entier);
var Milieu : entier ;
Début
    Si (Debut<Fin) Alors
        Milieu ← (Debut + Fin) div 2
        Tri(T, Debut, Milieu) ;
        Tri(T, Milieu + 1, Fin) ;
        Fusion(T, Debut, Milieu, Fin)
    Fin Si;
Fin;
```

2.5 Importance de l'ordre des appels récursifs

L'ordre d'un appel récursif par rapport aux autres instruction d'un algorithme influe considérablement sur le résultat obtenu. Les deux procédures suivantes en illustrent un exemple :

```
Procédure Afficher( n : entier);
Début
    Si (n ≥ 1) Alors
        Ecrire(n) ;
        Afficher(n-1) ;
    Fin Si;
Fin;
```

Résultat pour n = 5 : 5 4 3 2 1

```
Procédure Afficher( n : entier);
Début
    Si (n ≥ 1) Alors
        Afficher(n-1) ;
        Ecrire(n) ;
    Fin Si;
Fin;
```

Résultat pour n = 5 : 1 2 3 4 5

2.6 Types de récursivité

Selon le nombre d'appels et leur emplacement la récursivité peut être en plusieurs types :

2.6.1 Récursivité simple

La procédure ou la fonction s'appelle elle même une seule fois dans sans corps telle que la fonction $FactR$ dans l'exemple précédent.

2.6.2 Récursivité multiple

La récursivité est dite multiple si la procédure ou la fonction contient plus d'un appel récursif dans son corps telle que la fonction Fib dans l'exemple précédent.

2.6.3 Récursivité imbriquée

La récursivité est dite imbriquée si l'appel récursif contient un paramètre qui est aussi un appel récursif tel que :

$$\mathrm{F} \leftarrow \mathrm{F}(\mathrm{n},\, \mathrm{F}(\mathrm{n}))\,;$$

2.6.4 Récursivité croisée

La récursivité est dite croisée entre deux procédures (fonction) si chacune d'elle appelle l'autre tel que dans l'exemple suivant :

```
Fonction Pair( n : entier) : booleen;
Début
    Si (n = 0) Alors
        Pair ← Vrai ;
    Sinon
        Pair ← Impair(n-1) ;
    Fin Si;
Fin;

Fonction Impair( n : entier) : booleen;
Début
    Si (n = 0) Alors
        Impair ← Faux ;
    Sinon
        Impair ← Pair(n-1) ;
    Fin Si;
Fin;
```

Malgré qu'aucune fonction des deux ne fait appel à elle même, la récursivité existe. Elle est appelée aussi mutuelle.