

Chapitre 2 : La récursivité

Filière MIP – S2

Module : Informatique 2 : Algorithmique 2/Python

Pr. Badraddine AGHOUTANE

b.aghoutane@umi.ac.ma

A.U : 2023–2024

Plan

- Rappel
- **Les fonctions et les procédures en python.**
- La récursivité et son application dans des algorithmes.
- Les enregistrements et les fichiers en Python.
- La complexité des algorithmes et ses principaux types.
- Les preuves de correction et de terminaison d'un algorithme.

Résumé

Fonction **Nom_Fonction**(var:type,var:type): type
Variables internes (*Locales*)
Début
 Instructions

Retourner variable
Fin

Procédure **Nom_Procédure** (var:type, var:type)
Variables internes (*Locales*)
Début
 Instructions

 Instructions
Fin

Algorithme principale
Variables :(*Globales*)
Constantes:
Fonctions:
Début
 variable ← Nom_Fonction(var,var) } **Appel d'une Fonction**
 Ecrire(Nom_Fonction(var,var)) }
 Nom_Procédure(var,var) ← **Appel d'une Procédure**

Fin

Exercices d'application

Ecrire un **algorithme qui calcul le factoriel d'un nombre entier en utilisant une fonction** qui renvoie le **factoriel d'un nombre entier** :

Algorithme factoriel
Variable x, f : Entier
Fonction fact(x:entier):entier
Variable resultat, i: entier
Début
 resultat ← 1
 pour i de 1 à x faire
 resultat ← resultat * i
 finpour
retourner resultat
Fin

$n! = n(n-1)(n-2)...1$
 $0! = 1$ (by definition)
 $1! = 1$
 $2! = 2 \times 1 = 2$
 $3! = 3 \times 2 \times 1 = 6$

Début
 Ecrire ("Entrez un nombre entier x")
 Lire(x)
 f ← fact(x)
 Ecrire ("Le factoriel de x ", x, " est: ", f)
Fin

Exercices d'application

Ecrire un **algorithme qui fait appel** à une fonction qui permet de renvoyer la **puissance d'un nombre entier** :

$$a^n = \underbrace{a \times a \times \dots \times a}_{n \text{ facteurs}}$$

Algorithme Puissance

Variable x, n, pow : Entier

Fonction Puissance (x, : Entier, n : Entier) : Entier

Variable resultat, i : Entier

Début

Si n=0 alors

retourner (1)

Sinon

result ← 1

Pour i ← 1 à n **Faire**

result ← result * x

Finpour

retourner Result

Fin

Début

Ecrire ("Entrez un nombre entier x")

Lire(x)

Ecrire ("Entrez un exposant n")

Lire(n)

pow ← **Puissance**(x,n)

Ecrire ("La puissance =", pow)

Fin

Remarques :

La fonction qui calcul la **puissance** et celle qui renvoie le **factoriel** sont des **fonctions itératives** (*utilise des boucles*).

→ Pourquoi ne pas le faire de **manière récursive** ... plutôt que d'utiliser des **boucles...**?

C'est quoi la récursivité?

La **récursivité** est un concept très puissant : **décomposer un problème en un ou plusieurs sous-problèmes** qui sont de **même nature**, mais qui s'appliquent à un **nombre d'objets plus réduit**.

- $n! = n \cdot (n-1)!$ pour $n \geq 1$
- $X^n = X \cdot X^{n-1}$ pour $n \geq 1$

→ La **programmation récursive** sert à **remplacer les boucles**.

Récurtivité

Définition :

- Un sous-programme **A** peut appeler un autre sous-programme **B**.
- Lorsqu'un sous-programme **appelle lui-même** on parle d'**appel récursif**.
- **La récursivité** est la capacité d'un sous-programme (fonction ou procédure) à s'appeler lui-même.
- Elle permet de résoudre beaucoup de problèmes contenant des itérations complexes.
- Toute **méthode récursive** peut-être convertie en **méthode non-récursive**.
- Tout algorithme récursif devra contenir une **condition** (issue de secours) qui assure la **fin du nombre d'appels**.

Exemples

- Dans le **domaine végétal**, le romanesco (*hybride broccolo/ chou-fleur*) est très récursif
- Dans le **domaine animal**, un coquillage de nautilus
- **L'effet droste** (*image en abyme*).



Définition

- La **récursivité** permet de résoudre des problèmes complexes en les **décomposant en problèmes plus petits**
- Une **procédure (ou fonction)** est dite **récursive** lorsqu'elle fait **appel à elle-même**.
- La programmation récursive sert à remplacer les boucles (while, for, etc). Il faut vérifier si le processus ne boucle pas indéfiniment.
- Une **Procédure (ou Fonction)** est dite récursive si son exécution peut provoquer un ou plusieurs appels (dits **récurif**) à :
 1. Récursivité Simple
 2. Récursivité Multiple
 3. Récursivité Mutuelle
 4. Récursivité Imbriquée
 5. Récursivité Terminale et Non-Terminale

Exercices d'application

Calculer une factorielle?

1 Quel est le lien entre $5!$ et $4!$?

$5! = 5 \times 4!$ 

$5! = 5 \div 4!$

$5! = 5 - 4!$

$5! = 5 + 4!$

2 Quel calcul correspond à $10!$?

$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$ 

$10! = 10^{9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}$

$10! = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1$

$10! = 10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1$

Récurivité simple

Une **récurivité simple** contient un *seul appel récurif à la fonction F* dans le corps de la **fonction récurive F**.

→ Fonction qui s'invoque elle-même.

Exemple1: Calcul de la factorielle

- $n! = n(n-1)(n-2)\dots 2.1$ Si $n \geq 1$
- $n! = n(n-1)!$ pour $n \geq 1$

Noter que : $0! = 1! = 1$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

$$5! = 5 \cdot 4! \longrightarrow 4! = 4 \cdot 3! \longrightarrow 3! = 3 \cdot 2! \longrightarrow 2! = 2 \cdot 1! \longrightarrow 1! = 1 \cdot 0!$$

$$= 120 \longleftarrow = 24 \longleftarrow = 6 \longleftarrow = 2 \longleftarrow = 1$$

N.B : La **condition d'arrêt** (issue de secours) est $x=0$ ou $x=1$

Récurivité simple

Exemple1: Calcul de la factorielle

$$n! = n(n-1)(n-2)\dots 2.1 \quad \text{Si } n \geq 1$$

Noter que : $0! = 1! = 1$

$$n! = n(n-1)! \quad \text{pour } n \geq 1$$

```

Fonction fact(x:entier):entier
Variables resultat, i: entier
Début
  resultat ← 1
  pour i de 1 à x faire
    resultat ← resultat * i
  finpour
  retourner resultat
Fin
    
```

Fonction itérative (non recursive)

```

Fonction fact(x:entier):entier
Debut
  si x=0 alors // (ou si x=1)
    retourner (1)
  sinon
    retourner x*fact(x-1)
  Finsi
Fin
    
```

Fonction recursive qui s'invoque elle-même

$$5! = 5 \cdot 4! \longrightarrow 4! = 4 \cdot 3! \longrightarrow 3! = 3 \cdot 2! \longrightarrow 2! = 2 \cdot 1! \longrightarrow 1! = 1 \cdot 0!$$

$$= 120 \longleftarrow = 24 \longleftarrow = 6 \longleftarrow = 2 \longleftarrow = 1$$

N.B : La condition d'arrêt (issue de secours) est $x=0$ ou $x=1$

Récurivité simple

Comment ça marche ?

Trace pour $x = 3$:

```
Fonction fact(x:entier):entier
Debut
  si x=0 alors
    retourner(1)
  sinon
    retourner x*fact(x-1)
  Finsi
Fin
```

Appel à **fact (3)** :

3 ne vaut pas 0, donc je calcule $3 * \text{fact}(x-1)$ qui est $3 * \text{fact}(2)$:

2 ne vaut pas 0, donc je calcule $2 * \text{fact}(x-1)$ qui est $2 * \text{fact}(1)$:

1 ne vaut pas 0, donc je continue vers $1 * \text{fact}(0)$

0 vaut 0, donc **fact(0) renvoie 1**

donc **fact(1) renvoie $1 * \text{fact}(0) = 1 * 1 = 1$**

fact(2) renvoie $2 * \text{fact}(1) = 2 * 1 = 2$

fact(3) renvoie $3 * \text{fact}(2) = 3 * 2 = 6$

Condition d'arrêt

Récurivité simple

Comment ça marche?

Trace pour $n = 4$:

```
Fonction fact(x:entier):entier
Debut
  si x=0 alors
    retourner(1)
  sinon
    retourner x*fact(x-1)
  Finsi
Fin
```

Appel à **fact(4)**

. $4 * \text{fact}(3) = ?$

. Appel à **fact(3)**

.. $3 * \text{fact}(2) = ?$

.. Appel à **fact(2)**

... $2 * \text{fact}(1) = ?$

... Appel à **fact(1)**

.... $1 * \text{fact}(0) = ?$

.... Appel à **fact(0)**

.... **fact(0) renvoie la valeur 1**

.... $1 * \text{fact}(0) = 1 * 1 = 1$

... **fact(1) renvoie la valeur 1**

... $2 * \text{fact}(1) = 2 * 1 = 2$

.. **fact(2) renvoie la valeur 2**

.. $3 * \text{fact}(2) = 3 * 2 = 6$

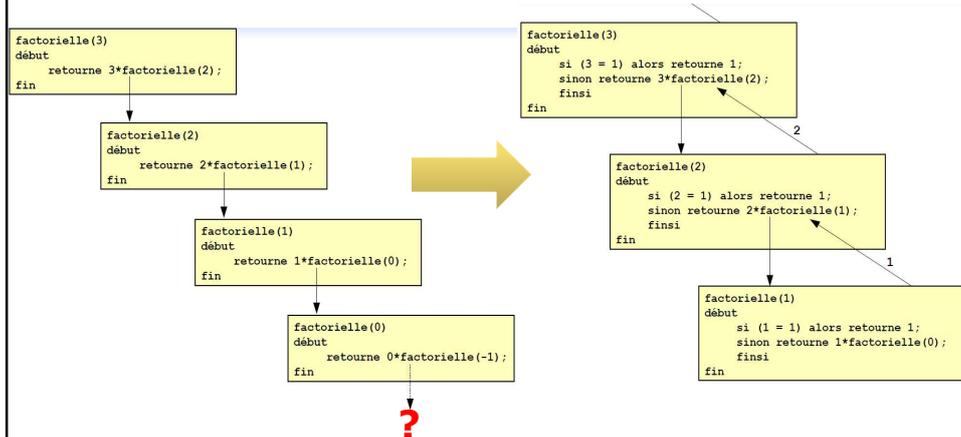
. **fact(3) renvoie la valeur 6**

. $4 * \text{fact}(3) = 4 * 6 = 24$

fact(4) renvoie la valeur 24

Récurivité simple

Tout algorithme récursif devra contenir une **condition** qui assure la **fin du nombre d'appels** ?.



→ On doit toujours tester en premier la **condition d'arrêt**, et ensuite, si la condition n'est pas vérifiée, lancer un appel récursif.

Calculer une factorielle à l'aide d'un programme python?

Soit n un entier naturel. Quel programme python permet de calculer $n!$?

```

n = int(input("Donner l'entier choisi.\n"))
factorielle = 1
for i in range(1,n+1):
    factorielle = factorielle * i
print(n,"! = ", factorielle)
    
```

```

n = int(input("Donner l'entier choisi.\n"))
factorielle = 1*2*...*n
print(n,"! = ", factorielle)
    
```

```

n = int(input("Donner l'entier choisi.\n"))
factorielle = 1
for i in range(1,n):
    factorielle = factorielle * i
print(n,"! = ", factorielle)
    
```

```

n = int(input("Donner l'entier choisi.\n"))
factorielle = 0
for i in range(1,n+1):
    factorielle = factorielle + i
print(n,"! = ", factorielle)
    
```

Calculer une factorielle à l'aide d'une fonction python?

```
def factorielle(n) :  
    P = 1  
    for i in range(n) :  
        P = P * i  
    return P
```

Fonction itérative (non réursive)

```
def factorielle(n) :  
    return n*factorielle(n-1)
```

Fonction réursive

```
def factorielle(n) :  
    L = list(range(1,n+1))  
    return sum(L)
```

```
def factorielle(n) :  
    if n == 0 :  
        return 1  
    else:  
        return n*factorielle(n-1)
```

Fonction réursive



Récurivité simple

Exercices d'application

- Écrire une fonction réursive qui calcule la puissance d'un nombre entier

Solution :

fonction puissance (x:entier, n:entier) : entier

Début

Si n=0

retourner 1

Si n=1

retourner x

Si n > 1

retourner x* puissance(x, n-1)

Fin

$$x^5 = x \times x^4$$

$\hookrightarrow x \times x^3$

$\hookrightarrow x \times x^2$

$\hookrightarrow x \times x^1$

$\hookrightarrow x \times x^0$

Récurivité simple

Exercices d'application

- Traduire la **fonction réursive puissance()** qui calcule la puissance d'un nombre entier

Solution :

```
def puissance(x:int,n:int):  
    if n==0 :  
        return 1  
    elif n==1:  
        return x  
    else :  
        return x*puissance(x,n-1)
```

```
x=int(input("Entrer un nombre entier : la base = "))  
n=int(input("Entrer un exposant = "))  
print("la puissance est: ", puissance(x,n))
```

```
Entrer un nombre entier : la base = 7  
Entrer un exposant = 2  
la puissance est: 49
```

Définition

- La **récurivité** permet de résoudre des problèmes complexes en les **décomposant en problèmes plus petits**
- Une **procédure (ou fonction)** est dite **réursive** lorsqu'elle fait **appel à elle même**.
- La **programmation réursive sert à remplacer les boucles** (while, for, etc). Il faut vérifier si le processus ne boucle pas indéfiniment.
- Une **Procédure (ou Fonction)** est dite réursive si son exécution peut provoquer un ou plusieurs appels (dits **réursif**) à :
 1. **Récurivité Simple**
 2. **Récurivité Multiple**
 3. **Récurivité Mutuelle**
 4. **Récurivité Imbriquée**
 5. **Récurivité Terminale et Non-Terminale**

Récurtivité multiple

Une **récurtivité** est **multiple** si il y a **plusieurs appels récurtifs à la fonction F dans le corps de la fonction récurtive F**.

→ Fonction qui s'invoque elle-même plusieurs fois

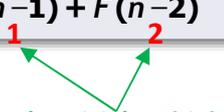
La **suite de Fibonacci** est définie par :

Suite de Fibonacci

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad , \text{ si } n > 1$$

Récurtivité multiple

Récurtivité multiple

La suite de **Fibonacci** est définie par (entier naturel) :

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

L'algorithme de **Fibonacci** s'écrit :

Fonction fib(n : entier) : entier

Début

Si (n = 0 OU n = 1) Alors

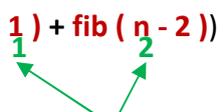
retourner (n)

Sinon

retourner (fib (n - 1) + fib (n - 2))

FinSi

Fin

Récurtivité multiple

Récurivité multiple

Exemple 2: suite de fibonacci

Équations de récurrences :

- $u(0) = 0, u(1) = 1$ (Base)
- $u(n) = u(n-1) + u(n-2), n \geq 1$ (récurrence)

```

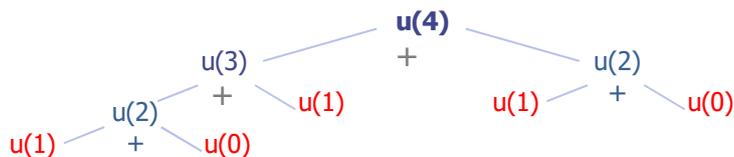
Fonction U(n:entier):entier
Debut
  si n=0 ou n=1 alors
    Retourner (n)
  sinon
    retourner U(n-1)+U(n-2)
Finsi
Fin
    
```

Condition d'arrêt

- Si l'argument n'est plus grand que 1, on retourne comme valeur n.
- Sinon, le résultat est $U(n-1) + U(n-2)$

Récurivité multiple

Exemple 2 (suite) : Voyons l'exécution : $u(4)$?



- La **condition d'arrêt** :
Si $(n=0$ ou $n=1)$ alors retourner (n)

```

Fonction U(n:entier): entier
Debut
  Si (n=0 ou n=1) alors
    retourner (n)
  Sinon
    retourner U(n-1)+U(n-2)
Finsi
Fin
    
```



```

Fonction U(n:entier): entier
Debut
  Si (n>1) alors
    retourner U(n-1)+U(n-2)
  Sinon
    retourner (n)
Finsi
Fin
    
```

Réversivité multiple : programme Python

La suite de **Fibonacci** est définie par :

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

Programme en langage Python

```
def fib(n:int):
    if(n==0 or n==1) :
        return n
    else:
        return fib(n-1)+fib(n-2)
```

Exécution : Saisir un nombre entier n : 6
La valeur de la suite de Fibonacci est: 8
PS C:\Users\Badraddine AGHOUTANE>

Réversivité Mutuelle

Définition :

Une **réversivité** est **mutuelle** ou croisée quand une **procédure P** (ou une fonction) appelle une **autre procédure Q** (ou une fonction) qui déclenche un appel récursif à la **procédure P**.

Remarque :

La situation est obligatoirement **symétrique**, puisque P déclenche un appel de Q, qui déclenche à son tour un appel de P.

Exemple :

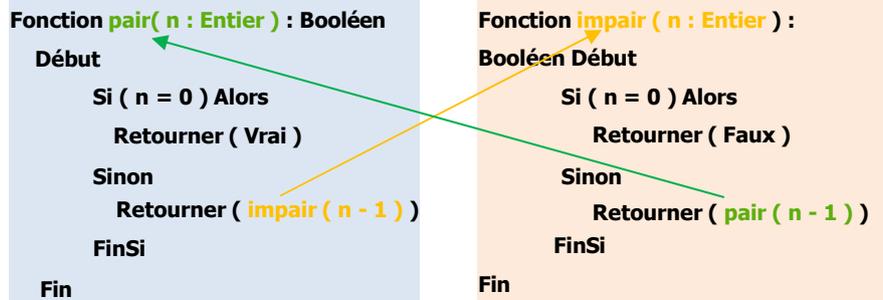
La **parité d'un entier naturel** peut être définie par :

$$\begin{aligned} \text{pair}(n) &= \begin{cases} \text{vrai} & \text{si } n = 0 \\ \text{impair}(n-1) & \text{sinon} \end{cases} \\ \text{impair}(n) &= \begin{cases} \text{faux} & \text{si } n = 0 \\ \text{pair}(n-1) & \text{sinon} \end{cases} \end{aligned}$$

La condition d'arrêt est n=0

Réversivité Mutuelle

Exemple : Les algorithmes correspondants s'écrivent :



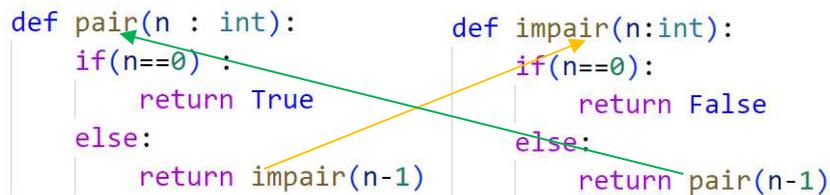
Les fonctions **pair()** et **impair()** s'invoquent **mutuellement** : **pair(3)**
pair(3)->**impair(2)**->**pair(1)**->**impair(0)** Condition d'arrêt n=0

La **dernière** invocation renvoie **Faux** :

impair(0)=Faux->**pair(1)=Faux**->**impair(2)=Faux**->**pair(3)=Faux**
 → Le nombre n'est pas **Pair** (Il est donc **Impair**)

Réversivité Mutuelle

Exemple : Les algorithmes correspondants s'écrivent :



```

n = int(input("Saisir un nombre entier n : "))
if (pair(n)==True) :
    print("La valeur" +str(n) + " est: pair")
else :
    print("La valeur" +str(n) + " est: impair")
  
```

Les fonctions **pair()** et **impair()** s'invoquent mutuellement (**Excécution**) :

```

Saisir un nombre entier n : 36
La valeur36 est: pair
  
```

```

Saisir un nombre entier n : 25
La valeur25 est: impair
  
```

Exemple :

La fonction **parité** qui calcule si un entier naturel est pair ou impair **sans utilisation de la récursivité** :

Programme Python

```
1 nombre = int(input("Entrez un nombre entier : "))
2 def parité(n):
3     if nombre % 2 == 0:
4         print(nombre, "est un nombre pair.")
5     else:
6         print(nombre, "est un nombre impair.")

nombre = int(input("Entrez un nombre entier : "))
parité (nombre)
```

Récursivité imbriquée

Définition :

- Une **récursivité** est **imbriquée** si *une fonction F (ou procédure récursive P) contient un appel imbriqué.*

Exemple :

- La fonction **d'Ackermann** (entiers naturels) est définie comme suit :

$$A(n, p) = \begin{cases} p + 1 & \text{si } n = 0 \\ A(n - 1, 1) & \text{si } n > 0 \text{ et } p = 0 \\ A(n - 1, \underbrace{A(n, p - 1)}) & \text{sinon} \end{cases}$$

Récurtivité imbriquée

Exemple :

L'algorithme de la **fonction d'Ackermann** correspondant s'écrit :

Fonction ackermann (n , p : Entier) : Entier

Début

Si (n = 0) Alors

Retourner (p + 1)

Sinon

Si (p = 0) Alors

Retourner ackermann (n - 1 , 1)

Sinon

Retourner ackermann (n - 1 , ackermann (n , p - 1))

FinSi

Fin

$$A(n, p) = \begin{cases} p + 1 & \text{si } n = 0 \\ A(n - 1, 1) & \text{si } n > 0 \text{ et } p = 0 \\ A(n - 1, A(n, p - 1)) & \text{sinon} \end{cases}$$

Récurtivité imbriquée

Programme en langage Python

Ecrire une fonction qui retourne la valeur calculée en utilisant la fonction d'**Ackermann**.

```
def ackermann(n: int, p:int):
    if(n==0) :
        return (p+1)
    elif (p==0):
        return (ackermann(n-1, 1))
    else:
        return (ackermann(n-1, ackermann(n, p-1)))

n = int(input("Saisir le premier paramètre de la fonction Ackermann(n) : "))
p = int(input("Saisir le deuxième paramètre de la fonction Ackermann(p) : "))
print("La valeur de la fonction ackermann est: ", ackermann(n,p))
```

L'exécution de la fonction **Ackermann()** :

```
Saisir le premier paramètre de la fonction Ackermann(n) : 3
Saisir le deuxième paramètre de la fonction Ackermann(p) : 2
La valeur de la fonction ackermann_est: 29
```

Réversivité terminale

Définition

Une définition de **fonction F** est **réursive terminale** quand tout appel récursif est de la forme **return F(...)**;

La **valeur retournée** est directement la valeur obtenue par un appel récursif, **sans qu'il n'y ait aucune opération sur cette valeur.**

somme(a,b)

Function **somme** (a, b: entier) : entier

Debut

Si (b=0) alors
retourner a

Sinon
retourner **somme** (a+1, b-1)

Finsi

Fin

somme(4,2) = somme(5,1) = somme(6,0) = 6

Réversivité terminale

Programme en langage Python

Ecrire en **langage Python** la fonction **somme** qui retourne la **somme de deux valeurs** sans aucune operation effectuée sur ces valeurs.

```
def somme(a : int, b:int):  
    if(b==0) :  
        return a  
    else :  
        return somme(a+1, b-1)  
  
a = int(input("Saisir le premier paramètre : "))  
b = int(input("Saisir le deuxième paramètre : "))  
print("La somme est: ", somme(a,b))
```

La fonction **somme()** est une fonction **recursive terminale** :

```
Saisir le premier paramètre : 5  
Saisir le deuxième paramètre : 8  
La somme est: 13
```

Récurtivité non terminale

Définition

Si le résultat de l'**appel récursif** est utilisé pour réaliser un traitement (fait partie d'une expression).

Exemple : Ecrire une fonction **somme** qui retourne la **somme de deux valeurs** sans aucune operation effectuée sur ces valeurs.

somme(a,b)

```
Fonction somme (a, b: Naturel): Naturel
  Debut
    si b=0 alors
      retourner a
    sinon
      retourner 1+somme(a,b-1)
    finsi
  Fin
```

somme(4,2) = 1+somme(4,1) = 1+1+somme(4,0) = 1+1+4=6

Récurtivité non terminale

Définition

L'appel récursif n'est pas la dernière instruction et/ou il n'est pas isolée (fait partie d'une expression).

```
def somme(a: int, b:int):
    if b==0 :
        return a
    else:
        return 1+somme(a,b-1)
a = int(input("Saisir le premier paramètre : "))
b = int(input("Saisir le deuxième paramètre : "))
print("La somme est: ", somme(a,b))
```

La fonction **somme()** est une fonction **recursive non terminale** :

```
Saisir le premier paramètre : 4
Saisir le deuxième paramètre : 2
La somme est: 6
```

$somme(5,7) = 1+somme(4,6) = 1+1+somme(4,5) = 1+somme(4,6) = \dots = 1+1+1+1+1+1+1+1+somme(4,0)=12$

La recette de récursivité

1. S'assurer que le problème peut se décomposer en un ou plusieurs **sous-problèmes de même nature**.
2. Identifier le **cas de base** qui est le plus petit problème qui ne se décompose pas en sous-problèmes
3. Résoudre(P) =
 - Si P est un cas de base,
le résoudre directement
 - Sinon
 - décomposer P en sous-problèmes P1, P2, P3, ...
 - résoudre récursivement P1, P2,...
 - combiner les résultats pour obtenir la solution pour P

Conclusions

- Les algorithmes récursifs permettent de résoudre des problèmes complexes.
- Il est sain de ramener un problème à des sous-problèmes.
- La récursivité permet de ramener un problème à un sous-problème qui est le problème lui-même avec des données plus simples.
- Les algorithmes récursifs sont le plus souvent plus gourmands en ressource que leurs équivalents itératifs.