جامعة مولاي إسماعيل
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵎⵓⵍⴰⵢ ⵉⵙⵎⴰⵄⵉⵍ
UNIVERSITÉ MOULAY ISMAÏL

كلية العلوم
ⵜⴰⵖⵍⵓⵍⵜ ⵏ ⵜⵣⵔⴼⵜ
FACULTÉ DES SCIENCES

Chapitre 2 : La récursivité

Filière MIP – S2

Module : Informatique 2 : Algorithmique 2/Python

Pr. Badraddine AGHOUTANE

b.aghoutane@umi.ac.ma

A.U : 2024–2025

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)
        Ecrire(Nom_Fonction(var,var))

        Nom_Procédure(var,var)
        … … … …
    Fin
```

Appel d'une Fonction

Appel d'une Procédure

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
```

$$n! = n(n-1)(n-2)\ldots 1$$

$0! \equiv 1$ (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** :

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

```
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** (*utilisant des boucles*).

Plutôt d'utiliser des **boucles**, pourquoi ne pas le faire de **manière récursive** ...?

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.

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

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

Récursivité

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.
- **La récursivité** permet de résoudre beaucoup de problèmes utilisant des itérations complexes.
- Toute **fonction récursive** peut-être convertie en **fonction itérative et vice versa**.
- 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/ choux- 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.
- On trouve plusieurs formes particulières de récursivité :

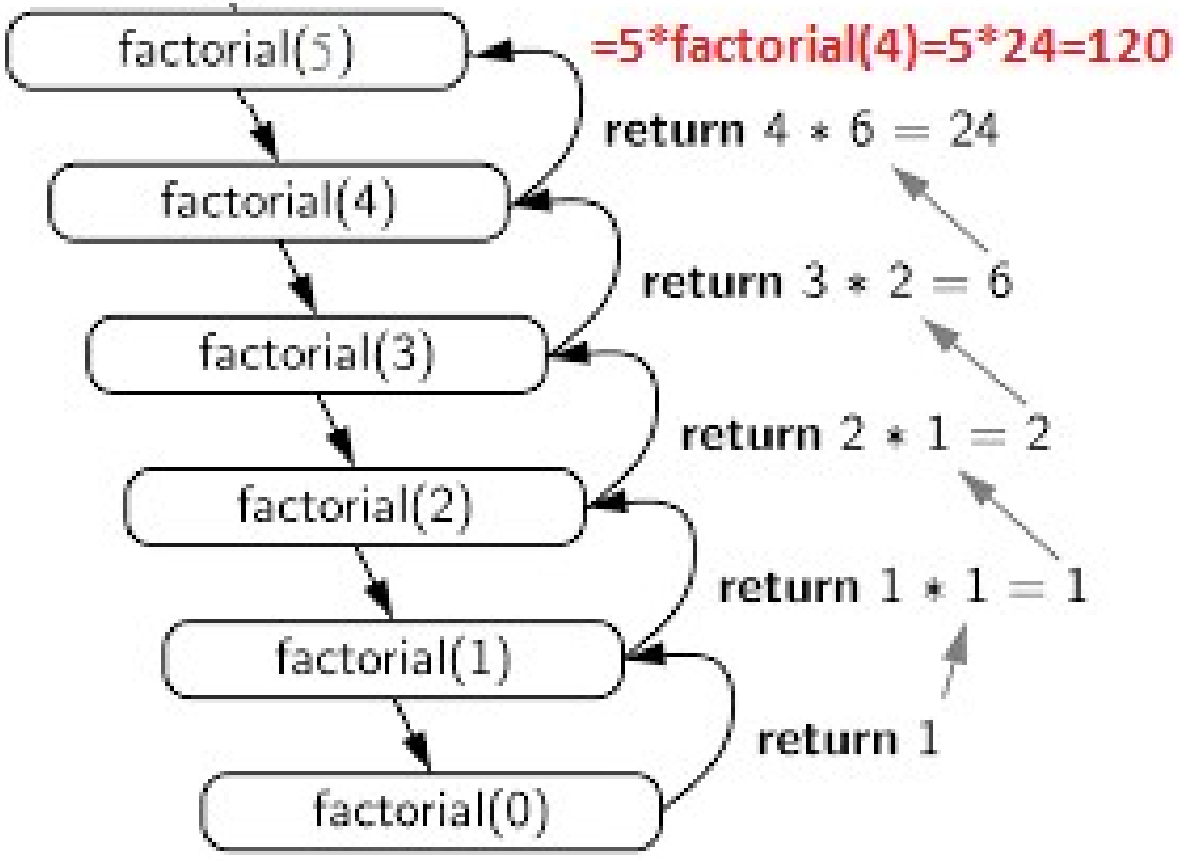
1. **Récursivité Simple**
2. **Récursivité Multiple**
3. **Récursivité Mutuelle**
4. **Récursivité Imbriquée**
5. **Récursivité Terminale et Non-Terminale**

Exercices d'application

Calculer une factorielle?

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

- [ ] $5! = 5 \times 4!$ ✓
- [ ] $5! = 5 \div 4!$
- [ ] $5! = 5 - 4!$
- [ ] $5! = 5 + 4!$



Récursivité simple

Une **récursivité simple** contient un ***seul appel récursif à la fonction F*** dans le corps de la **fonction récursive F.**

➔ **Fonction qui s'invoque elle-même.**

Exemple1: Calcul de la factorielle

- **n! = n(n-1)*(n-2)*…2*1** Si **n≥1**
- **n! = n*(n-1)!** pour **n≥1**

Noter que : **0! = 1! =1**

- **5! = 5 . 4 . 3 . 2 . 1 = 120**

- 5! = 5.4! ⟶ 4! = 4.3! ⟶ 3! = 3.2! ⟶ 2! = 2.1! ⟶ 1! = 1.0!

  = 120 ⟵ = 24 ⟵ = 6 ⟵ = 2 ⟵ = 1

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

Récursivité simple

Exemple1: Algorithme de calcul de la factorielle

1. n! = n(n-1)(n-2)…2.1 Si **n≥1**

Noter que : **0! = 1! =1**

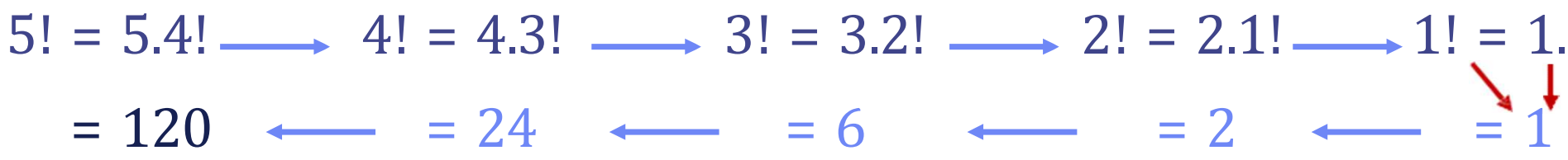
2. n! = n(n-1)! **pour n≥1**

```
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



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

Récursivité 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*fact(x-1)** qui est **3*fact(2)** :

2 ne vaut pas **0**, donc je calcule **2*fact(x-1)** qui est **2*fact(1)** :

1 ne vaut pas **0**, donc je continue vers **1*fact(0)**

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

donc **fact(1)** renvoie **1*fact(0)= 1*1=1**

fact(2) renvoie **2 * fact(1) = 2 * 1 = 2**

fact(3) renvoie **3 * fact(2) = 3 * 2 = 6**

Condition d'arrêt

Récursivité simple

Comment ça marche?

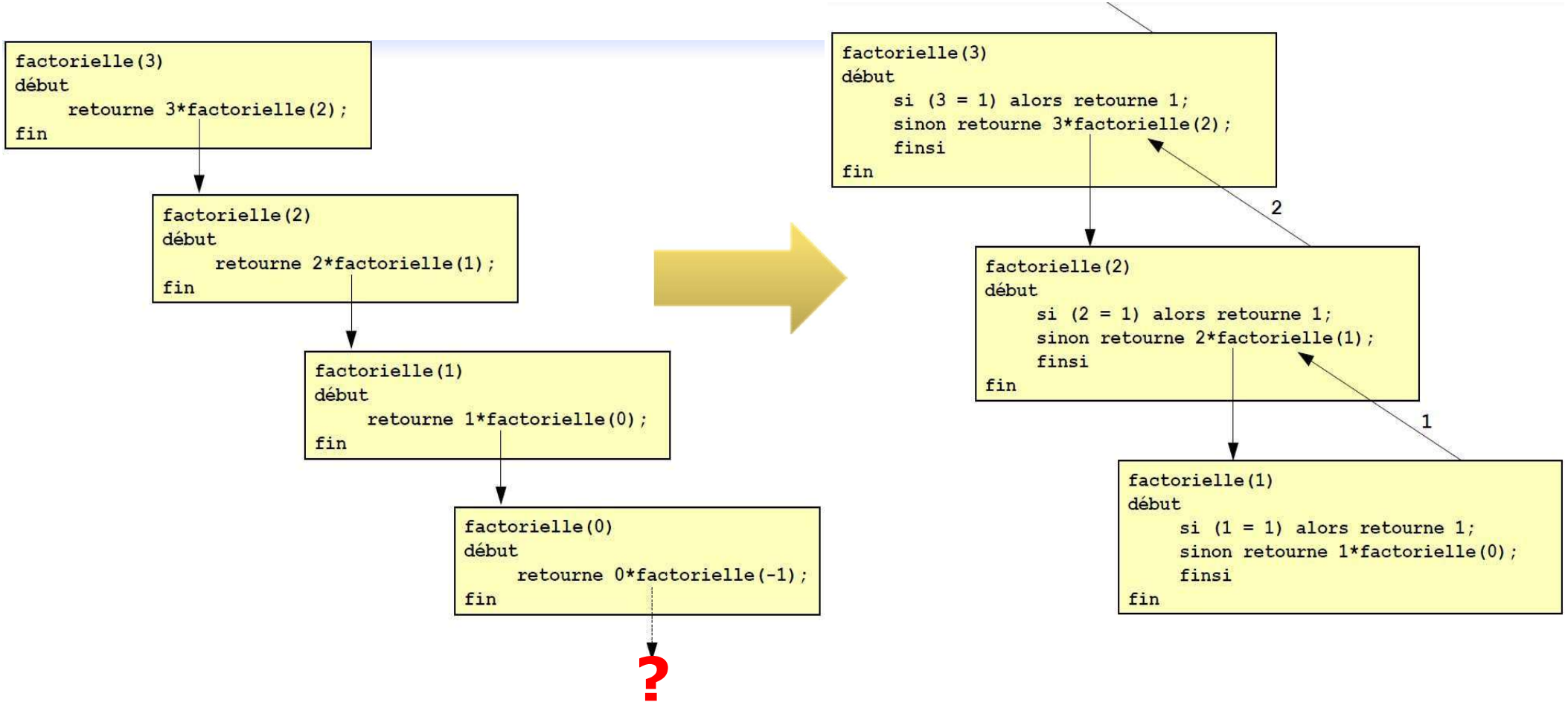
```
Fonction fact(x:entier):entier
Debut
        si x=0 alors
            retourner(1)
        sinon
            retourner x*fact(x-1)
        Finsi
Fin
```

Trace pour n =4 :

Appel à **fact(4)**
. 4*fact(3) = ?
. Appel à fact(3)
. . 3*fact(2) = ?
. . Appel à fact(2)
. . . 2*fact(1) = ?
. . . Appel à fact(1)
. . . . 1*fact(0) = ?
. . . . Appel à fact(0)
. . . . fact(0) retourne la valeur 1
. . . . 1*fact(0) = 1*1 = 1
. . . fact(1) retourne la valeur 1
. . . 2*fact(1) = 2*1 = 2
. . fact(2) retourne la valeur 2
. . 3*fact(2) = 3*2 = 6
. fact(3) retourne la valeur 6
. 4*fact(3) = 4*6 = 24
fact(4) retourne la valeur 24

Récursivité 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)
```



Fonction itérative

```
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 récursive)

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

Fonction récursive

```
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 récursive

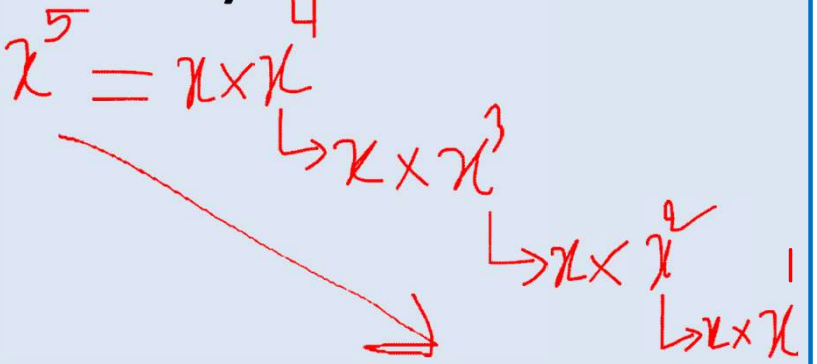
Récursivité simple

Exercices d'application :

- Écrire une fonction récursive 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
```



Récursivité simple

Exercices d'application

- Traduire la **fonction récursive 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
```

Récursivité multiple

Une **récursivité** est **multiple** si il y a ***plusieurs appels récursifs à la fonction F dans le corps de la fonction récursive F***.

➔ **Fonction qui s'invoque elle-même plusieurs fois**

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

Suite de Fibonacci

$$F(0) = 0$$
$$F(1) = 1$$
$$\mathbf{F(n) = F(n-1) + F(n-2)} \quad , \; si \; n > 1$$

Récursivité multiple

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

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

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
```

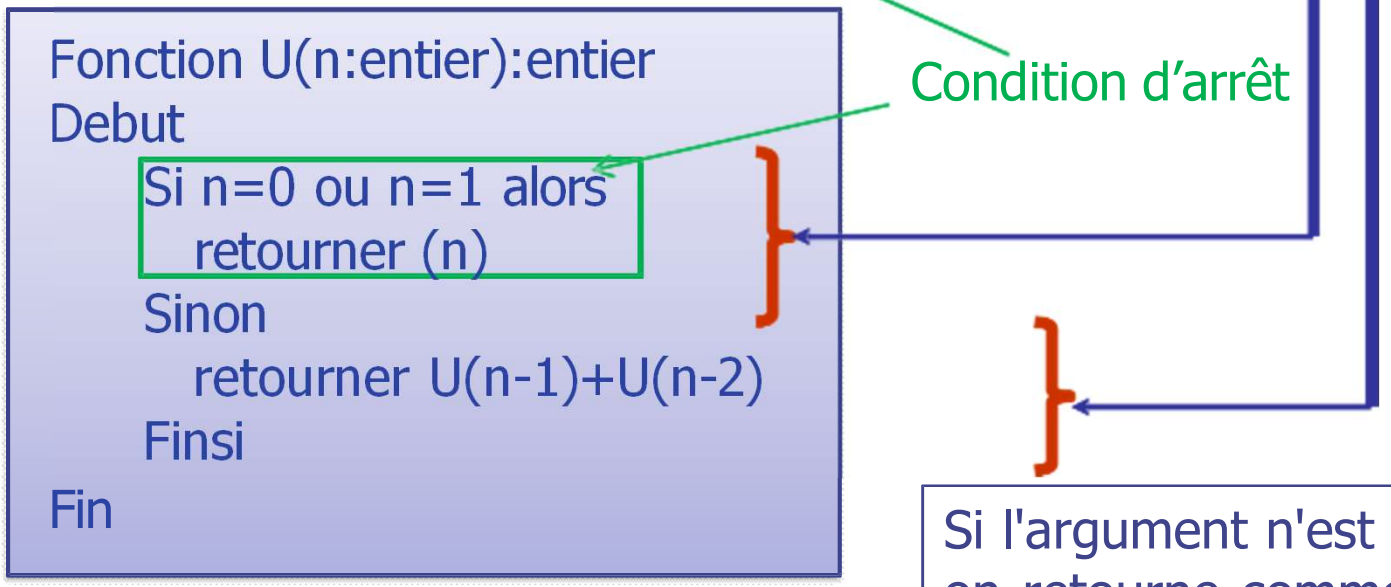
1 2

Récursivité 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>1 (récurrence)

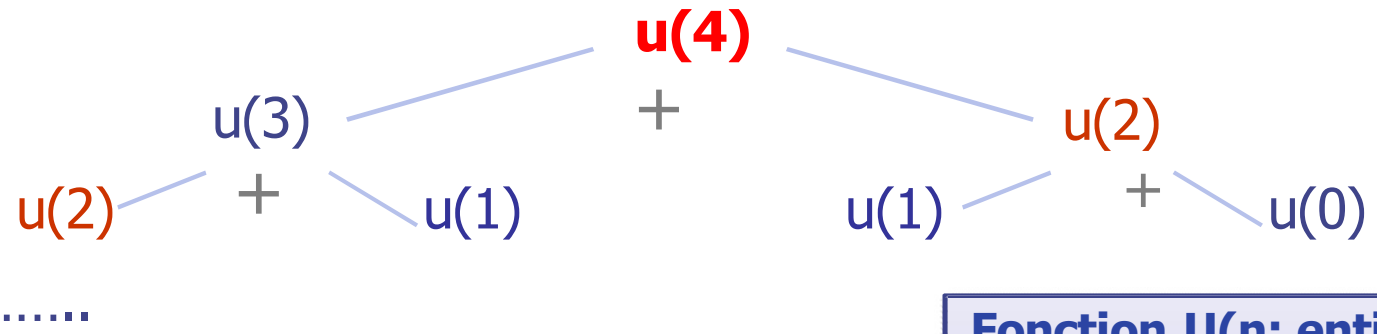


Si l'argument n'est plus grand que 1, on retourne comme valeur n. Sinon, le résultat est U(n-1) + U(n-2)

Récursivité multiple

- **Exemple 2 (suite)**

  Voyons l'exécution : **u(4)?**

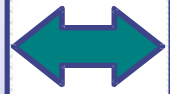


.....

- **La condition d'arrêt:**

  Si (n=0 ou n=1) **alors retourne (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
```

Récursivité 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)

n = int(input("Saisir un nombre entier n : "))
print("La valeur de la suite de Fibonacci est: ", fib(n))
```

Exécution :

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

Récursivité Mutuelle

- **Définition :**

Une **récursivité** est **mutuelle (*ou* *croisée*)** quand une fonction P appelle une function Q qui déclenche un appel récursif à P.

- **Remarque :**

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

- **Exemple :**

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

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

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

Récursivité Mutuelle

Exemple : Les algorithmes correspondants s'écrivent :

```
Fonction pair(n : Entier) : Booléen
   Début
        Si ( n = 0 ) Alors
           retourner ( VRAI )
        Sinon
           retourner ( impair ( n - 1 ) )
        FinSi
   Fin
```

```
Fonction impair(n : Entier) : Booléen
Début
        Si ( n = 0 ) Alors
            retourner ( FAUX )
        Sinon
            retourner ( pair ( n - 1 ) )
FinSi Fin
```

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

pair(3) → **impair(2)** → **pair(1)** → **impair(0)**

La dernière invocation renvoie Faux :

impair(0)=Faux → **pair(1)=Faux** → **impair(2)=Faux** → **pair(3)=Faux** → **Faux**

Récursivité Mutuelle

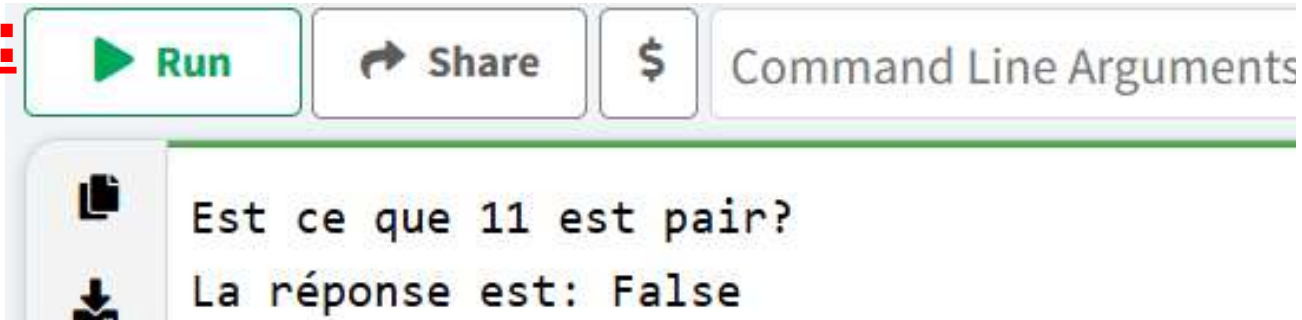
Exemple : Le programme python correspondant est:

py binaire.py pair.py +

```
def pair(n):
    if (n==0):
        return True
    else:
        return impair(n-1)
def impair(n):
    if (n==0) :
        return False
    else :
        return (pair(n-1))

print("Est ce que 11 est pair? \nLa réponse est:",pair(11))
```

Exécution :



Récursivité imbriquée

Définition :

Une **récursivité** est **imbriquée** si ***une procédure récursive P contient un appel imbriqué***

Exemple :

La fonction **d'Ackermann** (*entiers naturels*) est définie par :

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

Récursivité imbriquée

Exemple :

L'algorithme de la **fonction d'Ackermann** 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
    FinSi
Fin
```

Récursivité imbriquée

Exemple : Le programme python correspondant est :

.py binaire.py pair.py ackermann.py +

```
import sys
sys.setrecursionlimit(10000)# It sets recursion limit to 10000.

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

print("La fonction ackermann de 11, 10 est :", ackermann(3, 4))
```

Exécution :



Récursivité terminale

Définition :

Une définition de **fonction f** est **récursive 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)

```
Fonction somme (a, b : entier) : entier
    Début
        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

Récursivité terminale

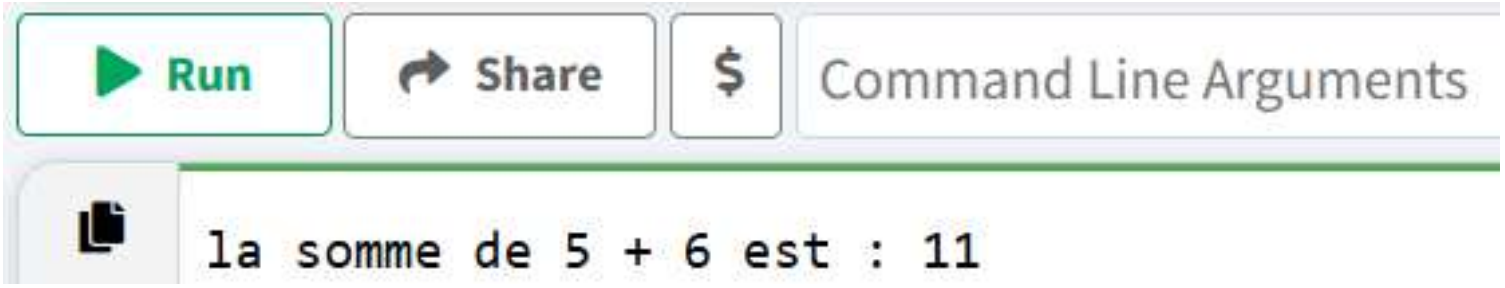
Exemple : Le programme python correspondant est :

py | binaire.py | pair.py | ackermann.py | somme.py

```
def somme (a,b):
    if (b==0):
        return a
    else:
        return somme(a+1, b-1)

print("La somme de 5 + 6 est :", somme(5,6))
```

Exécution :



Récursivité non terminale

Définition

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

somme(a,b)

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

Récursivité non terminale

Exemple : Le programme python correspondant est :

```
def somme (a,b):
    if (b==0):
        return a
    else:
        #print("Appel de la fonction 1+somme(",a,",",b,") = ",1+somme(a,b-1))
        return (1+somme(a, b-1))

print("La somme de 2 + 3 est :", somme(2,3))
```

Exécution :

