Récursivité

1 Pile des appels de fonction

Pour rappel un programme, mémorisé sur un disque dur, est chargé en **mémoire vive** pour être exécuté par le processeur.

Certains **registres** du processeur permettent au processeur de ne pas perdre le fil du calcul surtout s'il y a des **sauts**, des **branchements** et des **appels de fonction**.

Le registre EIP (Instruction Pointer) ou PC (Program Counter) contient l'adresse mémoire de la **prochaine** exécution à exécuter.

Ce pointeur (variable contenant une adresse mémoire) est initialisé à l'adresse de la première instruction du programme.

Nous avons vu l'année dernière le **cycle d'un processeur** au cours de l'exécution d'un programme.

1. **Lire** l'instruction référencée par le pointeur PC ou EIP.
2. Additionner la taille en octets de l'instruction à EIP (+4 ou +8).
3. **Exécuter** l'instruction lue à l'étape 1. Si l'instruction ne comporte pas de saut, de branchement ou d'appel de fonction, EIP "pointe" déjà sur l'emplacement en mémoire de l'instruction suivante à cause de l'étape2.
   Sinon EIP est modifié au cours de l'étape 3.
4. Retourner à l'étape 1

Que se passe-t-il lorsque dans un programme il y a des appels de fonctions ?

La possibilité d'écrire des fonctions dans un langage de programmation permet de factoriser le code (éliminer les redondances) et ainsi de rendre le code plus lisible et aussi plus sûr.

Cependant cela nécessite au niveau du processeur des mécanismes plus complexes que ceux utilisés pour un branchement ou un saut.

Observons avec PythonTutor l'exécution du programme suivant

```
def f1():
    i = 3
    print("dans f1, i = ",i)

i = 1
print("dans le main, i = ",i)
f1()
print("retour dans le main, i = ",i)
```

Nous observons que :

1. La variable nommée `i` dans la fonction `f1` n'existe qu'au cours de l'exécution de cette fonction. On dit que la portée de cette variable est **locale** à cette fonction.
   On dit que `i` est une variable locale de la fonction `f1`.

2. L'instruction `i = 1` dans le `main` définit une variable **globale**.

Observons avec PythonTutor l'exécution du programme suivant

```
def f1(i):
    print('le paramètre i vaut ',i)
    i = 3
    print("dans f1, i = ",i)

i = 1
print("dans le main, i = ",i)
f1(7)
print("retour dans le main, i = ",i)
```

Nous observons que :

1. Le **paramètre** `i` de la fonction `f1` est une variable **locale** à cette fonction. Ce paramètre est modifié par l'instruction `i = 3`
2. L'instruction `i = 1` dans le `main` définit une variable **globale**

Observons avec PythonTutor l'exécution du programme suivant

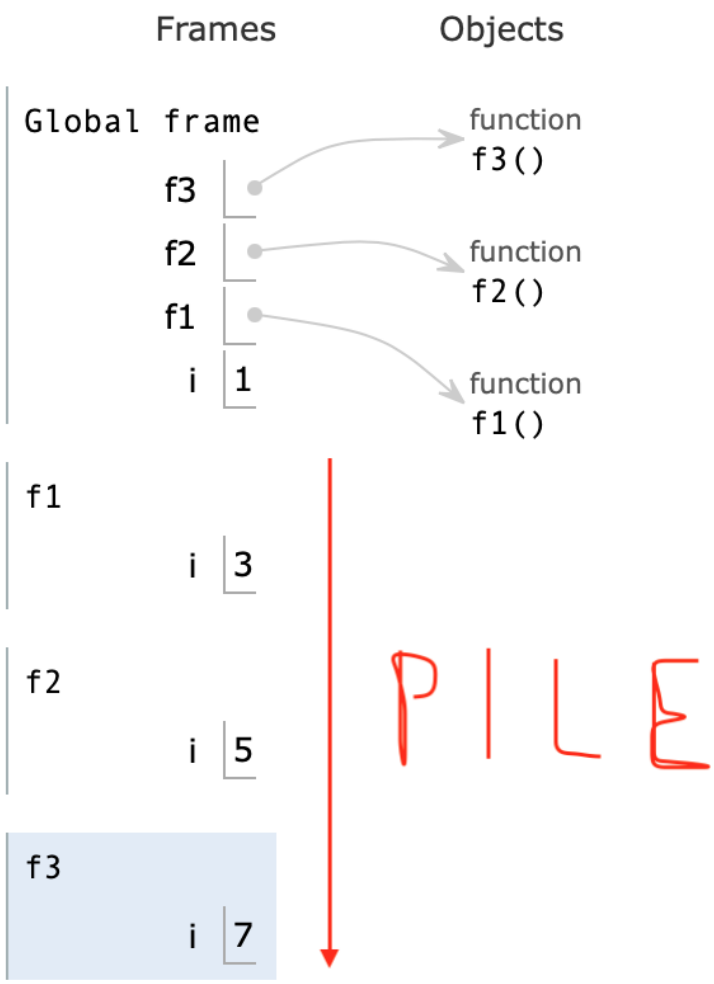
```
def f3():
    i = 7
    print("dans f3, i = ",i)

def f2():
    i = 5
    print("dans f2, i = ",i)
    f3()
    print("retour dans f2, i = ",i)

def f1():
    i = 3
    print("dans f1, i = ",i)
    f2()
    print("retour dans f1, i = ",i)

i = 1
print("dans le main, i = ",i)
f1()
print("retour dans le main, i = ",i)
```

Nous observons dans PythonTutor la pile des appels des fonctions.



En Informatique, une **pile** (stack) est une **structure de données** d'usage très fréquent, dont la caractéristique principale est que le premier élément placé sur la pile est le dernier à en sortir. On dit que c'est une LIFO (Last In, First Out). Imaginer une pile d'assiettes.

Cette structure de données est dynamique et évolue au cours du temps par l'intermédiaire uniquement de deux primitives, **empiler (push)** ou **dépiler (pop)**.

Lorsqu'une fonction est appelée, plusieurs informations formant ce qu'on appelle un **bloc d'activation** (activation block) est empilé.

A la fin de l'exécution de la fonction, le bloc d'activation est dépilé.

Chaque bloc d'activation contient les paramètres de la fonction, les variables locales de la fonction et **deux pointeurs qui permettent de remettre les choses en place dans leur état précédent**

1. Un pointeur contient l'adresse de l'instruction à exécuter à la fin de l'exécution de la fonction (adresse de retour)
2. Un pointeur permet de remettre la pile dans son état avant l'appel de la fonction (jusqu'à où faudra-t-il dépiler à la fin de l'exécution de la fonction)

2 Problème

On souhaite définir une fonction définie sur les entiers naturels , appelée factorielle et définie ainsi :

Factorielle de n est le produit de tous les entiers qui précèdent n

$n! = n \times (n-1) \times (n-2) \times ..... \times 2 \times 1$

3 Solution itérative

Pour calculer $n!$ on pourrait multiplier n par $n-1$ puis multiplier le résultat obtenu par $n-2$, et continuer jusqu'à multiplier le résultat par 2

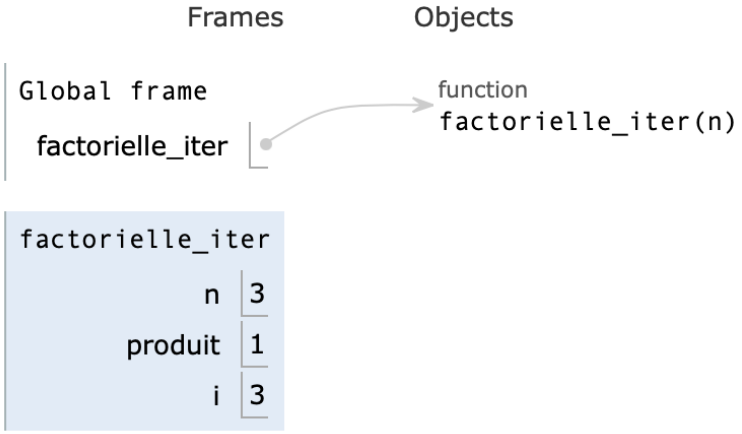
C'est ce que fait la fonction suivante

```
def factorielle_iter(n):
    produit = 1
    for i in range (n,1,-1):
        produit *= i
    return produit
```

Comment va être exécutée une telle fonction sur une valeur particulière par exemple `factorielle_iter(3)` ?

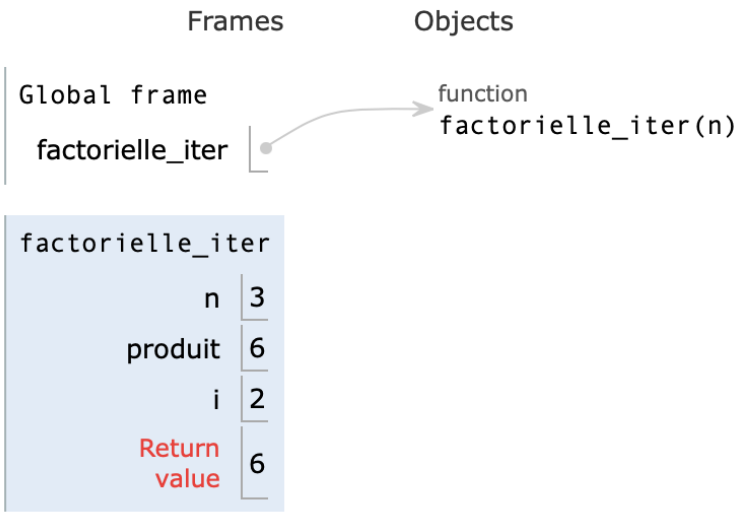
Utilisons Python Tutor pour avoir une idée de l'exécution

Sur le **segment de pile** de la mémoire vive est empilé le **bloc d'activation** de `factorielle_iter(3)` , représenté en bleu par la valeur du paramètre `n`, les valeurs des variables locales `produit` et `i`



Ne sont pas représentées les valeurs des registres du processeur au moment de l'appel de la fonction, ni la valeur de retour de la fonction

Cependant Python Tutor représente la valeur de retour à la fin de l'exécution de la fonction juste avant que le contexte d'exécution soit dépilé de la pile de la mémoire



4 Solution récursive

Mais en regardant de plus près on se rend compte que la définition de $n!$ **contient celle de** $(n-1)!$

$n! = n \times \underbrace{(n-1) \times (n-2) \times ..... \times 2 \times 1}_{(n-1)!}$

Autrement dit on peut définir factorielle n **par récurrence** :

En mathématiques vous avez vue en première des suites définies par récurrence, voici par exemple une suite géométrique particulière, celle des puissances de 2, définie par récurrence :

(condition initiale) $x_0 = 1$

(formule de récurrence) $x_n = 2 \times x_{n-1}$

De même on peut définir $n!$ par récurrence,

(condition initiale) $1! = 1$

(formule de récurrence) $n! = n \times (n-1)!$

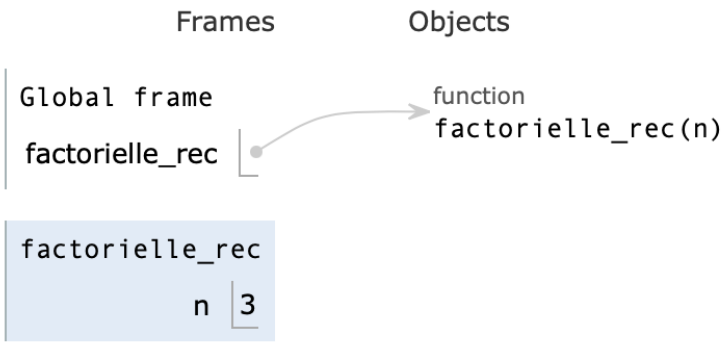
La définition en Python est quasiment la traduction de la définition mathématique par récurrence

```
def factorielle_rec(n):
    # condition initiale
    if n == 1:
        return 1
    # formule de récurrence
    else:
        return n*factorielle_rec(n-1)
```

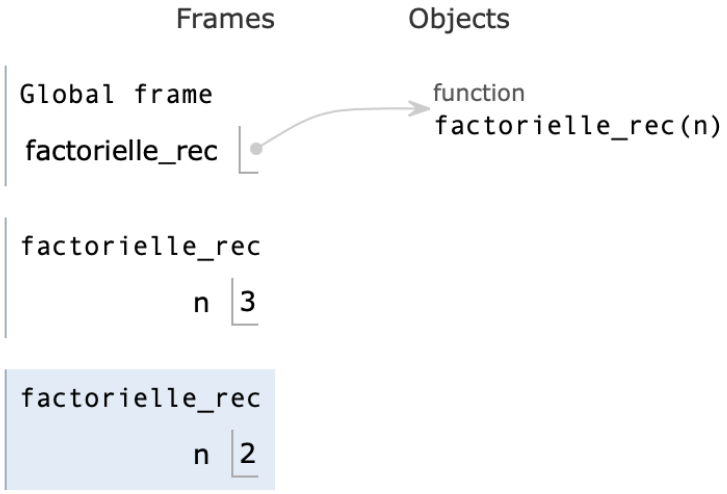
Comment va être exécutée une telle fonction sur une valeur particulière par exemple `factorielle_rec(3)` ?

Utilisons Python Tutor pour avoir une idée de l'exécution

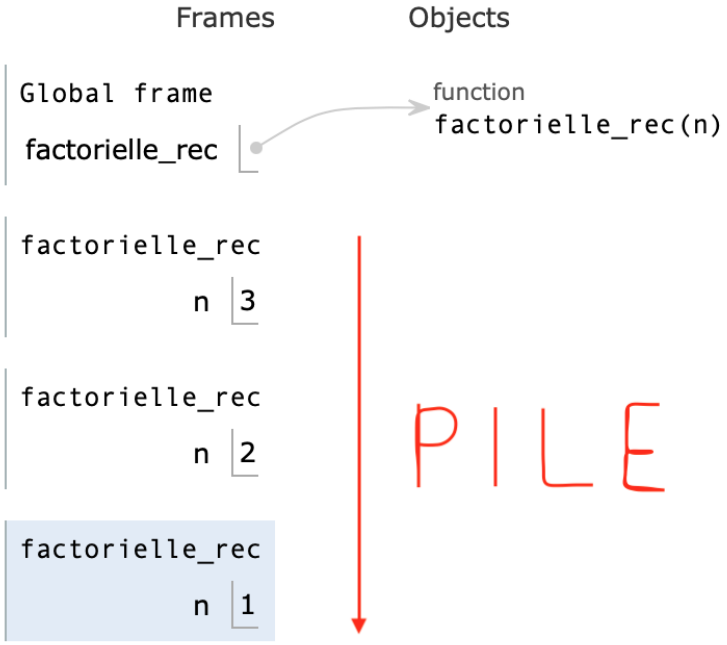
Sur le **segment de pile** de la mémoire vive est empilé le **bloc d'activation** de `factorielle_rec(3)` ,



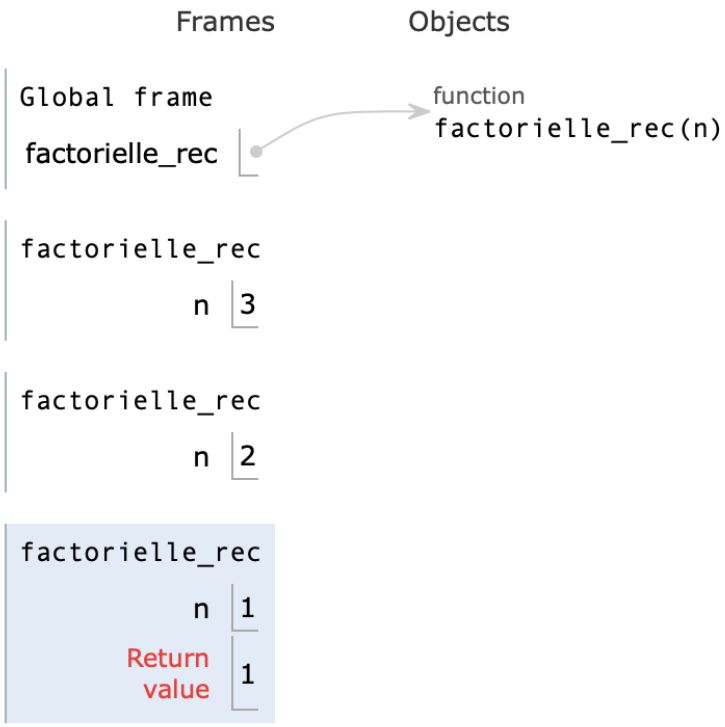
Puisque dans le corps de `factorielle_rec(3)`, est appelé `factorielle_rec(2)`, le **bloc d'activation** de `factorielle_rec(2)` est empilé (la croissance de la pile se fait vers le "bas")



Puisque dans le corps de `factorielle_rec(2)`, est appelé `factorielle_rec(1)`, le **bloc d'activation** de `factorielle_rec(1)` est empilé



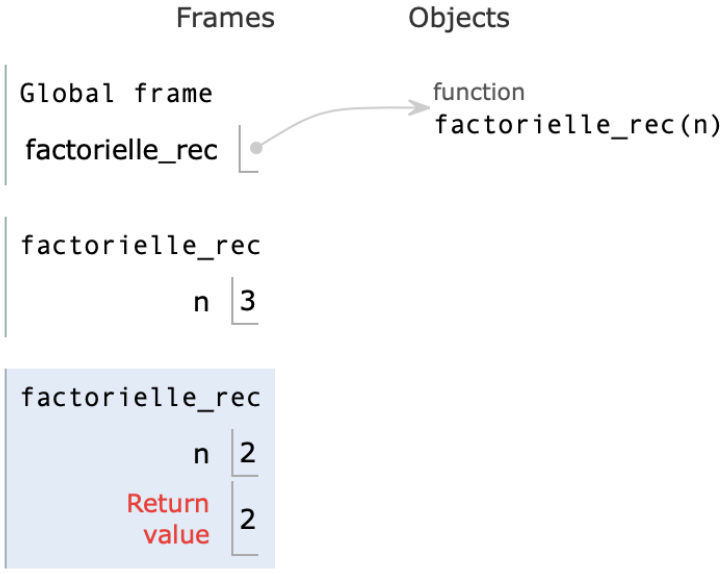
Maintenant



A l'exécution de `factorielle_rec(1)`, la valeur de retour est 1 et le bloc d'activation de `factorielle_rec(1)` est **dépilé** et

La valeur de retour 1 remplace `factorielle_rec(1)` dans le bloc d'activation de `factorielle_rec(2)` et la valeur de retour de `factorielle_rec(2)` est :

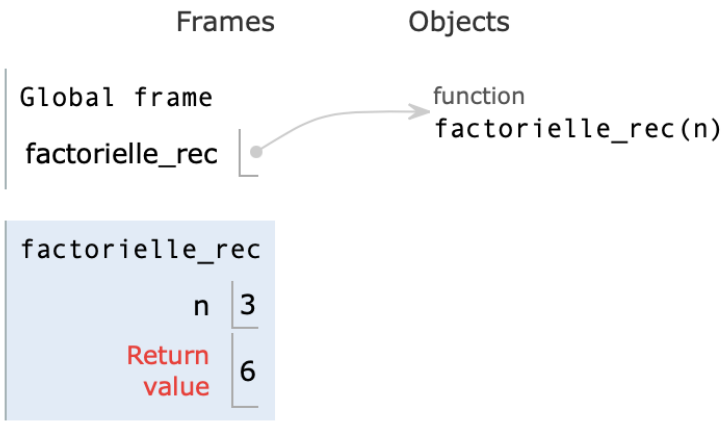
`2* factorielle_rec(1) = 2`



A son tour le bloc d'activation de `factorielle_rec(2)` est dépilé

La valeur de retour 2 remplace `factorielle_rec(2)` dans le bloc d'activation de `factorielle_rec(3)` et la valeur de retour de `factorielle_rec(3)` est :

`3* factorielle_rec(2) = 6`



Finalement le bloc d'activation de `factorielle_rec(3)` est dépilé

ATTENTION ! La hauteur de la pile des appels récursifs est limitée ce qui peut être un inconvénient si dans un problème il y aura un grand nombre (au delà de 1000) appels récursifs

On aura alors le message d'erreur suivant :

```
RecursionError: maximum recursion depth exceeded in comparison
```

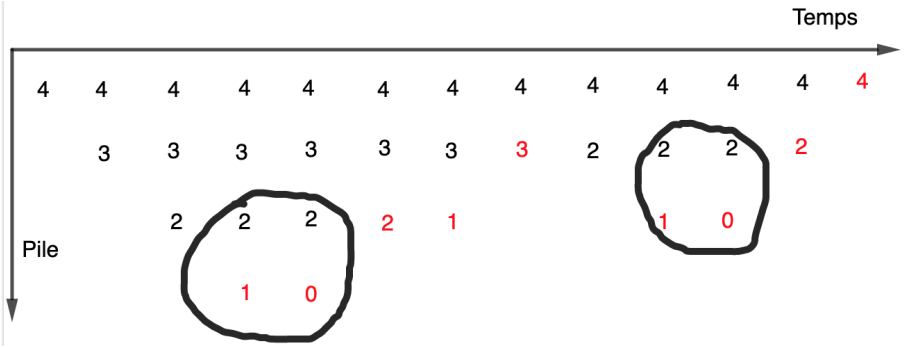
5 La suite de Fibonnacci

La suite de Fibonnacci F_n est défini par récurrence par
$F_0 = 0$ et $F_1 = 1$ (cas de base)
$F_n = F_{n-1} + F_{n-2}$ (récurrence) pour $n \geqslant 2$
On peut alors définir une fonction `fib(n)` **récursive**

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

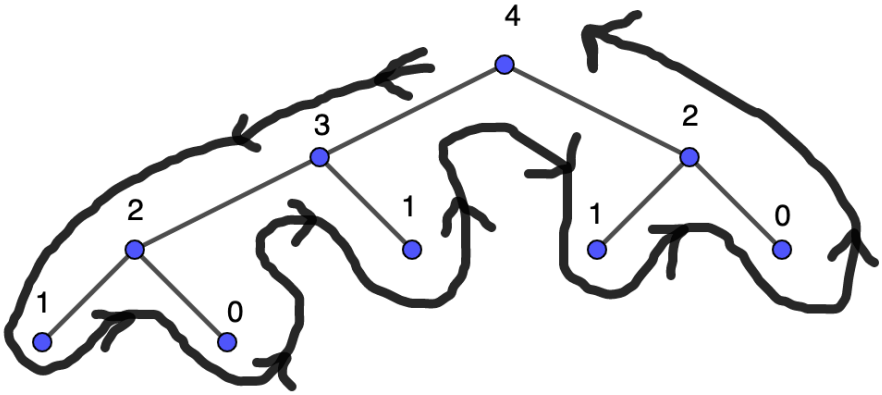
Que se passe-t-il si on exécute `fib(4)` ?

On a représenté la pile des appels récursifs en fonction du temps comme dans PythonTutor



Les valeurs en rouge sont les valeurs où il y aura une valeur de retour

On peut aussi utiliser un arbre pour représenter la suite des appels et des retours, on a mis le sens de parcours de l'arbre (en profondeur)



Dans tous les cas on observe que des appels sont répétés ce qui **ralentit** l'exécution de `fib(n)` si n est "grand"(voir TP)

6 La fonction f_{91}

Cette fonction est définie par récurrence par :
Si $n > 100$ alors $f_{91}(n) = n - 10$
Sinon $f_{91}(f_{91}(n + 11))$
On défini alors la fonction **récursive**

```
def f_91(n):
 if n > 100:
     return n - 10
 else:
     return f_91(f_91(n + 11))
```

On montre que $f_{91}(n) = 91$ si $n \leqslant 101$

7 La fonction d'Ackermann

La fonction d'Ackermann a est définie sur des entiers naturels x et y ainsi
$a(0, y) = y + 1$ pour $y \in \mathbb{N}$
$a(x, 0) = a(x - 1, 1)$ pour $x \geqslant 1$
$a(x, y) = a(x - 1, a(x, y - 1))$ pour x et y plus grands que 1

```
def a(x,y):
 if x == 0:
     return y + 1
 if y == 0:
     return a(x-1,1)
 return a(x-1,a(x,y-1))
```

A retenir :
En Informatique on rencontre beaucoup de situations **définies par récurrence** (une condition initiale + une formule de récurrence)
On peut traduire en Python ces définitions par récurrences en **fonctions récursives** où dans le corps de la fonction il y a un ou plusieurs appels de la fonction mais sur des arguments de taille plus petite de telle sorte qu'à un moment la suite des appels tombe sur un **cas de base**
La longueur de la suite des appels est limité à 1000

8 Exercices

Ex 1

```
def f(i,j):
    i = 1
    j = 2
    print(id(i)," ",id(j))
    k = i + j
    return k
# main
i = 3
j = 4
print(id(i)," ",id(j))
print(f(i,j))
```

1. Quels sont les **paramètres** de f ? Quelles sont les variables **locales** de f ?
2. Quelles sont les variables **globales** ?
3. Rechercher sur le Web le rôle de la fonction Python `id()` ?
4. Quel est le résultat de l'instruction `print(f(i,j))` ?

Ex 2

Compléter le texte suivant.

"Pour gérer intelligemment l'ordre des appels de fonctions dans un programme on utilise une structure de données appelée .....

La **dernière** fonction appelée en exécution, une fois son exécution terminée, sera la .... à quitter cette structure de donnée. Aussi on utilise le sigle .... pour caractériser cette structure de données"

Ex 3

On veut écrire une fonction `echanger(i,j)` qui échange le contenu des variables `i` et `j` en procédant ainsi

```
def echanger(i,j):
    temp = j
    j = i
    i = temp
# main
i = 2
j = 3
```

```
echanger(i,j)
print("i = ",i)
print("j = ",j)
```

On constate que ça ne marche pas. Pourquoi ?

Ex 4

Observer avec Python Tutor que cette fois ci, la fonction `echanger` dont les paramètres sont des tableaux contenant une seule valeur, fait bien le travail demandé.

```
# t1 et t2 sont des tableaux contenant un seul élément
def echanger(t1:list,t2:list)->None:
    temp = t1[0]
    t1[0] = t2[0]
    t2[0] = temp


t1 = [2]
t2 = [3]
print(f"Le contenu de t1 est {t1[0]}, celui de t2 est {t2[0]}")
echanger(t1,t2)
print(f"Le contenu de t1 est {t1[0]}, celui de t2 est {t2[0]}")
```

Ex 5

Quelle est la différence entre

```
def factorielle_rec(n):
    # condition initiale
    if n == 1:
        return 1
    # formule de récurrence
    else:
        return n*factorielle_rec(n-1)
```

et

```
def factorielle_rec(n):
    # condition initiale
    if n == 1:
        return 1
    # formule de récurrence
    return n*factorielle_rec(n-1)
```

Ex 6

Exécuter avec Python Tutor les fonctions `fib(4)` et $f_{91}(99)$ et observer la pile des appels récursifs.

Ex 7

Dérouler les appels récursifs et les retours pour l'exécution de `f(4)` où `f` est définie récursivement par :

```
def f(n):
    if n == 0:
        return 1
```

```
if n == 1:
    return 2
return f(n-2)*f(n-1)
```

Ex 8

Beaucoup de propriétés mathématiques sont définies par **récurrence** :
Par exemple

1. a et b étant des entiers naturels :
   pgcd(a,0) = a
   pgcd(a,b) = pgcd(b,r) tel que $a = bq + r$ avec $0 \leqslant r < b$
2. Le coefficient binomial $\binom{n}{p}$ est défini pour n et p entiers naturels par
   $\binom{n}{p} = 1$ si $p = 0$ ou $n = p$
   Sinon $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$

Transformer les propriétés précédentes en fonctions récursives

Ex 9

Ecrire une fonction récursive en Python `longueur(liste:list)->int` qui renvoie le nombre d'éléments contenus dans la liste où

1. A la fin de l'exécution de cette fonction la liste est vide
2. A la fin de l'exécution de cette fonction la liste est inchangée

Ex 10

Soit u_n la suite définie par récurrence par :
$u_0 = 2$ et $u_{n+1} = \dfrac{4u_n - 2}{u_n + 1}$
Ecrire une fonction récursive `u(n)` qui renvoie le terme de rang n de la suite ci-dessus.

Ex 11

Ecrire une fonction récursive en Python `max(liste:list)` qui renvoie la valeur du plus grand nombre contenu dans la liste.

Ex 12

La suite de Sylvester est définie par récurrence par
$u_0 = 7$ et $u_n = (u_{n-1} - 1) \times u_{n-1} + 1$

1. Ecrire une fonction récursive `sylv(n)` qui renvoie le terme de rang n de la suite de Sylvester
2. $u_1 = 43$, $u_2 = 1807$, $u_3 = 3263443$ or si on calcule la somme des chiffres de ces termes de manière répétée jusqu'à ce qu'on ait un nombre inférieur ou égal à 9 on obtient 7.
   Vérifier le
3. On admettra que "calculer la somme des chiffres d'un nombre n de manière répétée jusqu'à ce qu'on ait un nombre inférieur ou égal à 9" revient à calculer n % 9
   Ecrire une fonction `verif_conjecture(n)` qui renvoie vrai si $n\%9 == 7$ et faux sinon.
   Utiliser cette fonction pour vérifier la conjecture pour plusieurs termes de la suite. (On démontre par récurrence que cette propriété est toujours vraie)

Ex 13

Soit la suite récurrente (u_n) définie par
$u_2 = 2$ et $u_{n+1} = f(u_n)$
où f est définie sur $]0, +\infty[$ par $f(x) = \frac{1}{2}(x + \frac{2}{x})$

Cette suite converge vers $\sqrt{2}$, autrement dit cette suite permet de calculer des valeurs approchées de $\sqrt{2}$

1. Définir en Python une fonction `f(x)` qui retourne l'mage de x par la fonction f
2. Définir une fonction **récursive** `u(n)` qui retourne le terme d'indice n da la suite (u_n) pour $n \geqslant 2$

Ex 14

Une chaîne de caractères (en minuscules et ne contenant pas d'espaces) est un palindrome si en la lisant dans les deux sens on a la même suite de caractères.

Ecrire une fonction récursive `est_palindrome(chaine:str)->bool` qui renvoie vraie si la chaîne est un palindrome et faux sinon.

Par exemple

```
>>> est_palindrome('laval')
True
>>> est_palindrome('1664')
False
>>> est_palindrome('esoperesteicietserepose')
True
```

Ex 15

Ecrire une fonction récursive en Python `retourner(liste:list)->list` qui renvoie une nouvelle liste contenant les éléments de `liste` mais dans l'ordre inverse.

Ex 16

Ecrire une fonction récursive en Python `trier(liste:list)->list` qui renvoie une nouvelle liste contenant les éléments de `liste` triés dans l'ordre croissant.

Ex 17

1. Définir une fonction récursive `nombre_de_chiffres(n)` qui retourne le nombre de chiffres de l'entier naturel n

   Par exemple `nombre_de_chiffres(1234)` retourne 4
2. Simuler l'état de la pile pour l'appel de `nombre_de_chiffres(1234)`

Ex 18

1. Définir une fonction récursive `nombre_de_diagonales(n)` qui retourne le nombre de diagonales du polygone régulier à n côtés
2. Simuler l'état de la pile pour l'appel de `nombre_de_diagonales(6)`

Ex 19

Dans une liste `L` on a une suite d'entiers naturels.

Si deux éléments de cette liste d'indice `i` et `j` avec `i < j` sont tels que `L[i] > L[j]` on dit qu'il y a **une** inversion dans la liste `L`

Par exemple : si la liste L est [1,4,2,3] alors dans cette liste il y a deux inversions :

1. L[1] et L[2] forment une inversion car $1 < 2$ et L[1] $>$ L[2]

2. L[1] et L[3] forment une inversion car $1 < 3$ et L[1] $>$ L[3]

Définir une fonction récursive `nombre_inversions(L)` qui renvoie le nombre d'inversions de la liste `L`

Par exemple

```
>>> nombre_inversions([1,5,4,0])
3
```

Ex 20

BAC
Ex 2 Métropole Sujet 0 2021

Ex 21

Ecrire une **fonction récursive** `ajouter_fin(liste,val)` qui prend en paramètre une liste chaînée et une valeur et qui renvoie une nouvelle liste où la valeur a été insérée dans une cellule à la fin de la liste passée en paramètre.

```
class Cellule:
    """
    définit un élément d'une liste chainée"""
    def __init__(self,v,s):
        self.valeur = v
        self.successeur = s

    def __str__(self):
        return str(self.valeur)

def ajouter_fin(liste,val):
    pass

class Liste:
    """
    """
    def __init__(self):
        self.tete = None

    def ajouter_tete(self,val):
        self.tete = Cellule(val,self.tete)

    def ajouter_fin(self,val):
        self.tete = ajouter_fin(self.tete,val)
```