# bilan TP7

A chaque instruction, l'état de la mémoire change.
1) Que se passe-t-il  lors de l'appel d'une fonction ou lors d'une déclaration de variable ?

x=3 (x est un entier qui est un type simple)
Un case mémoire est alors réservée à x dans laquelle on y met 5 (en binaire selon une norme bien précise)

x=[3,2] (x est une liste qui est un type structuré)
Dans ce cas, une case mémoire est réservée à x dans laquelle contient  l'adresse du début de l'espace mémoire pour coder la liste [3,2] (pour simplifier : l'adresse de la case qui contient le premier élément de la liste)
Autrement dit la valeur contenue dans la case mémoire réservée à x est une adresse, celle de son premier élément
On parle de pointeur ou de référence.

On comprend alors que dans ce programme :
x=2
y=x
y=3
print(x)
print(y)

y change mais pas x

Par contre dans celui-ci :
x=[2,3]
y=x
y[0]=0
print(x)
print(y)

x et y changent
Voici les captures d'écran

```
2
3
>>>
[0, 3]
[0, 3]
>>>
```

2) Maintenant, que se passe-il avec les fonctions ?
x=3
f(x)
print(x)

Dans cet exemple, au moment de l'instruction f(x), qui est un appel à f avec comme valeur du paramètre, la valeur de x, un espace mémoire est alors réservé dans lequel :
il y a d'abord une recopie de x, puis les instructions du corps de f sont exécutées.
A la fin, tout cet espace est libéré et on passe à l'instruction suivante.
Or f a pour effet de mettre à zéro la valeur du paramètre, donc met à zéro la « recopie » de x.

C’est cette recopie qui change, et qui « disparaît » à la fin de l’exécution de f(x), mais x ne change pas !

y=[3,5]
g(y)
print(y)
Dans le cas de g, c’est pareil, il y a recopie de la valeur de y qui est  donc l’adresse où se trouve le premier élément de y.
On met alors ce premier élément à zéro, donc à cet endroit de la mémoire le contenu change.
A la fin, quand g se termine, afficher y consiste à afficher les éléments de la liste y à l’adresse indiquée.
Or à cette adresse le contenu du premier élément a bien changé.
Autrement dit y change.
Voilà le programme et la capture d’écran :

```
def f(a):
   a=0
def g(t):
   t[0]=0
x=3
y=[3,8]
f(x)
g(y)
print(x)
print(y)
```

```
3
[0, 8]
>>>
```

3) Et dans le cas des fonctions récursives ?
A chaque appel de la fonction, un nouvel espace est réservé, il a donc une sorte d’empilement des appels successifs.
fact(4) fait à appel à fact(3) etc.
Lors de l’appel à fact(1), il y a une valeur retournée qui est 1.
fact(1) se termine puis fact(2), qui attendait cette valeur, retourne 2x1=2, ainsi fact(2) se termine.
etc.
L’instruction conditionnelle
if n≤1 :
  return 1
permet d’arrêter le processus d’empilement et de commencer celui de dépilement qui libère au fur et à mesure l’espace mémoire.
Il faut bien sûr s’assurer que cette condition « n≤1 » deviendra vraie à un moment donné.

Dans le cas de Fibonnacci :

```
def fibo(n):
   if n<=1:
      return 1
   else:
      return fibo(n-1)+fibo(n-2)


print(fibo(6))
```

```
13
```

Remarquez, que les suites récurrentes se programment aisément récursivement :
$u_0=0$
et pour tout n $u_{n+1}=u_n +2n-1$

```
def suite(n):
    if n==0:
        return 0
    else:
        return suite(n-1) +2*(n-1)-1


for i in range(6):
  print(suite(i))
```

```
0
-1
0
3
8
15
>>>
```

On peut même généraliser :
$u_0=ini$
et pour tout n $u_{n+1}=au_n +bn + c$

```
def suitegen(n,ini,a,b,c):
    if n==0:
        return ini
    else:
        return a*suitegen(n-1,ini,a,b,c) +b*(n-1) +c


for i in range(6):
  print(suitegen(i,0,1,2,-1))
```

```
0
-1
0
3
8
15
>>>
```