TP n° 11 – Variants et complexité

I Factorielle

On définit la factorielle d'un entier n par les relations : $0! = 1$ et $(n+1)! = (n+1)n!$
On propose le programme de calcul de $n!$ suivant où n est un entier strictement positif :

```
p = 1
c = 0
for c in range(1,n+1):
  p= c * p
```

Listing 1: Programme de calcul de la fonction factorielle.

Exercice 1. Recopier ce code dans un script idle. Le modifier afin d'afficher `c` et `p` à chaque étape de la boucle. Donner alors la relation entre ces deux variables.

Solution 1.

```
p = 1
c = 0
for c in range(1,n+1):
  p= c * p
  print(c,p)
```

On constate que à chaque étape, $p = !c$.

Exercice 2. Peut-on alors trouver ici un variant ou un invariant de boucle ? Si oui lequel ? Que peut-on en conclure sur ce script ? Justifier votre réponse.

Solution 2. c est strictement croissant, entier, majoré par n : c'est un variant de boucle. Donc le programme se termine.
On constate que à chaque étape, $p = c!$.

- L'invariant de boucle est la proposition $p = c!$.
- À l'initialisation, on a : $p = 1$, $c = 0$ et $1 = 0!$, la proposition est donc vraie.
- À la première itération dans la boucle : $c = 1$, $p = c * p = 1 * 1 = 1$ et $1 = 1!$, la proposition est donc vraie.
- On suppose donc $p_i = c_i!$ vraie. À la (i+1)^e itération : $c_{i+1} = c_i + 1$, $p_{i+1} = c_{i+1} * p_i = (c_i + 1) * c_i! = c_{i+1}!$. La proposition est donc toujours vraie dans la boucle.
- À la terminaison de la boucle, on a $c = n$ et donc finalement $p = n!$ ce qui prouve la correction de l'algorithme.

II Terminaison

```
a = 17
b = 7
while a >= b:
  a = a - b
```

Listing 2: Programme inconnu

Exercice 3. Recopier ce code dans un script idle. Le modifier afin d'afficher `a` et `b` à chaque étape de la boucle. En déduire le but de cet algorithme.

Solution 3.

```
a = 17
b = 7
while a >= b:
  a = a - b
  print(a,b)
```

$(a = 17, b = 7)$; $(a = 10, b = 7)$; $(a = 3, b = 7)$. La valeur finale de a est 3. Le programme renvoie le reste de la division euclidienne de a par b (`a % b` en python).

Exercice 4. Peut-on alors trouver ici un variant ? Si oui lequel ? Que peut-on en conclure sur ce script ? Justifier votre réponse.

Solution 4. Un variant de boucle est a.
- Initialement $a = 17$ et $b = 7$ donc $a \geq b$ et la boucle commence.
- On a comme condition de boucle $a \geq b$ donc on a toujours $a \geq 0$ dans la boucle.
- D'autre part, si on note a_+ la valeur de a à la fin d'une itération et a_0 sa valeur au début d'une itération, on a $a_+ = a_0 - b < a_0$ car $b > 0$.
  a est donc une variable strictement décroissante dans la boucle.
- a est donc entier, positif et strictement décroissant dans la boucle : la boucle termine.

III Complexités

En mathématiques, la suite de **Fibonacci** est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent.
Notée (F_n), elle est définie par $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$.
Le code suivant permet de calculer le n$^{\text{ème}}$ coefficient de la liste par une méthode récursive.

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

Listing 3: Fibonacci récursive

Exercice 5. Proposer une fonction `f_i(n)` qui renvoie le même résultat que `f_r(n)` sans être récursive.

Solution 5.

```
def f_i(n):
    if n==0:
        return 0
    s1,s2=0,1
    for i in range(2,n+1):
        g=s2
        s2=s1+s2
        s1=g
    return s2
```

Exercice 6. Exécuter `f_i(40)` puis `f_r(40)`, que constatez-vous ?

Solution 6. Le résultat est le même mais `f_r(40)` est beaucoup plus long à calculer que `f_i(40)`.

Afin d'analyser le constat de la question précédente, la suite va proposer de mettre en évidence la complexité des deux fonctions.

Exercice 7. Modifier la fonction `f_i(n)` afin qu'elle retourne un le résultat mais aussi un compteur qui indique le nombre d'opérations nécessaires pour obtenir le résultat.

Solution 7.

```
def f_i(n):
    if n==0:
        return 0,1
    s1,s2=0,1
    c_i=3
    for i in range(2,n+1):
        g=s2
        s2=s1+s2
        s1=g
        c_i+=4
    return s2,c_i
```

Exercice 8. La même manipulation est elle possible pour la fonction `f_r(n)` ? Pourquoi ? Justifier la nécessité de l'utilisation d'une variable **globale**.

Solution 8. Ce n'est pas possible car la fonction `f_r(n)` étant récursive, le compteur utilisé pour `f_i(n)` ne se propagerait pas au travers de la récursivité. Il faut utiliser une fonction globale car elle pourra être utilisée par toutes les instances de la fonction.

Rappel : Les variables globales se déclarent en Python comme une variable locale mais à l'extérieur d'une fonction.
Pour utiliser une variable globale sans modifier sa valeur, il suffit de l'appeler dans une fonction comme vous le feriez avec une variable locale. Les variables globales sont dangereuses en développement car l'on peut faire des erreurs en les manipulant (cf. Annexe).
Si la fonction doit modifier la valeur de la variable globale, alors il faut la déclarer en tant que telle dans la définition de la fonction en la précédant du mot-clé `global` comme dans l'exemple suivant.

```
def fonction():
        global variable_globale
        variable_globale=0
        ...
```

Exercice 9. Proposer une solution pour déterminer le nombre d'opérations nécessaires pour obtenir le $n^{ème}$ coefficient de la suite de Fibonacci grâce à la solution `f_r(n)`.

Solution 9.

```
c_r=0
def f_r(n): # récursif
    global c_r
    if n==0 :
        c_r+=1
        return 0
    if n==1 :
        c_r+=1
        return 1
    else :
        c_r+=1
        return f_r(n-1) + f_r(n-2)
f_r(10)
print(c_r)
```

Afin de constater l'évolution de la complexité, on souhaite créer deux listes `lci` et `lcr`, qui contiennent respectivement les valeurs des compteurs précédent pour les fonctions `f_i(n)` et `f_r(n)`, pour `n` allant de `1` à `N`.

Exercice 10. Proposer un script permettant de générer les deux listes `lci` et `lcr`.

Solution 10.

```
lci=[]
lcr=[]
n=10

for i in range(1,n):
    print(i)
    res_i=f_i(i)
    lci.append(res_i[1])
    c_r=0
    print(res_i[0],f_r(i))
    lcr.append(c_r)
```

Exercice 11. Tracer les valeurs de `lci` et `lcr`, en fonction de `n`, pour $N = 10$.

Solution 11.

```
import matplotlib.pyplot as plt

plt.plot(range(1,n),lci,range(1,n),lcr)
plt.show()
```

Exercice 12. Entre les deux algorithmes `lci` et `lcr`, la complexité de l'un est linéaire (c'est-à-dire en $\mathcal{O}(n)$), l'autre est de complexité exponentielle (donc en $\mathcal{O}(e^n)$). En vous aidant du tracé précédent, déterminer la complexité de chacun.
Donner une démonstration dans le cas de la complexité linéaire.

Solution 12. On a tracé le nombre d'opérations de chaque algorithme en fonction de n.
Pour `lci`, on obtient une droite : la complexité est linéaire.
Pour `lcr`, on obtient une courbe qui suit la focntion exp : la complexité est exponentielle.
Dans la fonction `lci`, on trouve une boucle `for` avec $n - 2$ itérations. Donc la complexité est $\mathcal{O}(n)$.

Exercice 13. Après avoir déterminé de `f_i(n)` et `f_r(n)` laquelle est la plus efficace, proposer une fonction `liste_fibonacci(n)` qui permet de générer la liste des `n` premiers termes de la suite. Afficher le résultat pour les 20 premiers termes.

Solution 13.

```
def liste_fibonacci(n):
    result=[]
    for i in range(n):
        result.append(f_i(i)[0])
    return result

print(liste_fibonacci(20))
```

Annexe : variable locale et variable globale

Une variable locale d'une fonction est une variable interne à une fonction et qui n'est connue que par la fonction : le reste du script ne « connaît » pas cette variable. Une variable globale est une variable qui est connue par la totalité du script. Par défaut toutes les variables qui sont à l'intérieur d'une fonction sont des variables locales à cette fonction : cela peut conduire à des comportements inattendus, notamment si on tente de modifier une variable globale à l'aide d'une fonction.

Le code suivant :

```
def fonction():
    var = var + 1
    print("valeur dans la fonction :", var)

var = 3

print("valeur avant l'appel de la fonction :", var)
fonction()
print("valeur après l'appel de la fonction :", var)
```

Affiche une erreur dans la fonction :
`UnboundLocalError: local variable 'var' referenced before assignment`
La variable `var` n'est pas connue par la fonction : elle ne peut pas la manipuler.

Le code suivant :

```
def fonction():
    var = 2
    print("valeur dans la fonction :", variable)

var = 3

print("valeur avant l'appel de la fonction :", variable)
fonction()
print("valeur après l'appel de la fonction :", variable)
```

Affiche :
`valeur avant l'appel de la fonction : 3`
`valeur dans la fonction : 2`
`valeur après l'appel de la fonction : 3`
Pour l'interpréteur python, la variable `var` déclarée dans la définition de la fonction est une variable locale qui n'est pas la même que celle à déclarée l'extérieur de la définition de la fonction : il les traite indépendamment l'une de l'autre. Une fois l'appel de la fonction terminer, la variable `var` n'a pas été modifiée par le script.

Le code suivant :

```
def fonction():
    global var
    var = 2
    print("valeur dans la fonction :", var)

var = 3
print("valeur avant l'appel de la fonction :", var)
fonction()
print("valeur après l'appel de la fonction :", var)
```

Affiche :
`valeur avant l'appel de la fonction : 3`
`valeur dans la fonction : 2`
`valeur après l'appel de la fonction : 2`
La variable `var` est déclarée comme globale dans la définition de la fonction : elle est modifiée par la fonction avec un effet pour la globalité du script.