# Cours 2 : Corrigé des exercices

## Exercice 1

— La fonction renvoie le quotient et le reste par la division euclidienne de $a$ par $b$.
— Elle est fiable si $a$ et $b$ sont des entiers, et $b$ non nul.
— En utilisant des noms de fonction et de variables plus explicites, en ajoutant une documentation de fonction et des commentaires.

## Exercices 2, 3 & 5

**Exemple 1.** *Division euclidienne*

```
def fonction_LOL(a, b) :
    q = 0
    r = a
    while r >= b :
        q = q + 1
        r = r - b
    return (q, r)
```

— La précondition est $a, b \in \mathbb{N} \times \mathbb{N}^\star$.
— La postcondition est $a = b \cdot q + r$ avec $q, r \in \mathbb{N} \times \mathbb{N}$.
— Un variant de boucle est $r$.
— Un invariant de boucle est $P_k$ : "$a = q_k \cdot b + r_k$".

**Exemple 2.** *Plus Grand Commun Diviseur*

```
def PGCD(a, b) :
    '''Recherche du PGCD de deux entiers a et b'''
    u = a
    v = b
    while u != v :
        if u > v :
            u = u - v
        else :
            v = v - u
    return (u)
```

**Remarque 1.** *pgcd(a,b) est noté* $a \wedge b$.

— La précondition est $a, b \in \mathbb{N}^\star \times \mathbb{N}^\star$.
— La postcondition est $u = a \wedge b$.
— Un variant de boucle est $u + v$ ou $\max(u, v)$.
— Un invariant de boucle est $P_k$ : "$u_k \wedge v_k = a \wedge b$".

**Exemple 3.**

```
# Un programme a tester
c = 0
while p > 0 :
    if c == 0 :
        p = p - 2
        c = 1
    else :
        p = p + 1
        c = 0
```

— La précondition est $p \in \mathbb{N}^\star$.
— La postcondition est $p = 0$.
— Un variant de boucle est $2 \cdot (p + c) + c = 2 \cdot p + 3 \cdot c$.

## Exercice 4

```
# Suite de Syracuse
def Syracuse(n) :
    '''fonction affichant les termes de la suite de Syracuse
    d'un entier n strictement positif.'''
    Sn = [n] # nombre de depart
    while Sn[-1] != 1 :
        # tant que le terme de la suite est different de 1
        # la terminaison est une conjecture qui, en depit de la simplicite de son enonce, \
        # n'a toujours pas ete demontree ni infirmee...
        if Sn[-1] % 2 : Sn.append(3 * Sn[-1] + 1)
        # si sn est impair, on le remplace par 3 * sn + 1
        else : Sn.append(Sn[-1] // 2)
        # sinon, on le divise par 2
    return (Sn)
```

## Exercice 6

```
# Calcul du nieme terme de la suite definie par U_n = sqrt{n + U_n-1}
def terme(a, n):
    '''renvoie le terme Un de la suite Un = sqrt{n + U_n-1} avec u0 = a.
    a est donc un flottant et n un entier.'''
    res = a
    for i in range(1, n + 1):
        # si n == 0, le programme ne parcourt pas la boucle for,
        # sinon, il la parcourt jusque n compris
        res = pow(i + res, 0.5)
        # l'invariant de cette boucle est que
        #res = U_i Ã  la fin de chaque iteration
    return res
```

## Exercice 7

```
def produit(a, b):
    '''renvoie le produit des entiers compris entre a et b compris.
    a et b sont donc des entiers tels que a < b.'''
    res = 1
    for i in range(a, b + 1):
        # i prendra donc les valeurs entieres successives entre a et b compris
        res *= i
        # l'invariant de cette boucle est res est le produit des i entiers
        # successifs commencant par a
    return res
```

## Exercice 8

```
# version criticable mais souvent rencontree (et toleree en IPT)
def test_occ1(seq, elt):
    '''renvoie True si elt appartient a la sequence seq, False sinon.'''
    for i in range(len(seq)):
        if elt == seq[i]:
            return True
        # l'invariant de boucle est "elt n'est pas dans les i premiers elements
        # de la sequence seq" a la fin de l'iteration
    return False

def test_occ2(seq, elt):
    '''renvoie True si elt appartient a la sequence seq, False sinon.'''
    n = len(seq)
    i = 0
    while i < n and not (elt == seq[i]) : # Attention ordre du test
        # le variant est n - i
        i += 1
        # l'invariant de boucle est "elt n'est pas dans les i premiers elements
        # de la sequence seq" a la fin de l'iteration
    return i < n and elt == seq[i] # idem
```

## Exercice 9

```
def approx(x, n):
    '''renvoie une approximation de la serie entiere Somme{n = 0 a + inf}{x^n}
    pour x compris entre 0 et 1 non compris'''
    res = 0
    for i in range(n + 1): # Attention, n compris !
        res += pow(x, i)
        # l'invariant de boucle est "res = somme{0 a i}{x^n}"
    return res
```

## Exercice 10

```
from math import factorial

def f_exp1(x, n):
    '''renvoie une approximation de exp(x) par la somme des n premiers termes
    de sa serie entiere.
    x est un flottant et n un entier.'''
    res = 0
    for i in range(n + 1):
        res += pow(x, i) / factorial(i) # math.factorial() est importe
        # l'invariant de boucle est "res = somme{n = 0 a i}{x^n/n!}"
    return res

def f_exp1bis(x, n):
    '''renvoie une approximation de exp(x) par la somme des n premiers termes
    de sa serie entiere.
    x est un flottant et n un entier.'''
    res = 1
    fac = 1 # fac permet de memoriser au fur et a mesure i! pour eviter de
    # le recalculer a chaque iteration
    for i in range(1, n + 1):
        fac *= i
        res += pow(x, i) / fac
        # l'invariant de boucle est "res = somme{n = 0 a i}{x^n/n!}"
    return res
```

## Exercice 11

```
def somme1(n):
    '''version recursive simple'''
    if n == 1 : # cas de base
        return 1
    else :
        return n + somme1(n - 1)

def somme2(n, acc = 0):
    '''version recursive terminale avec argument par defaut'''
    if n == 0 :
        return acc
    else :
        return somme2(n - 1, acc + n)

def somme2bis(n):
    '''version recursive terminale avec fonction auxiliaire'''
    def aux(i, acc):
        if i == 0:
            return acc
        else:
            return aux(i - 1, acc + i)
    return aux(n, 0)
```

## Exercice 12

```
# Calcul des termes d'une suite definie par recurrence
def Un(n) :
    '''Fonction qui permet de calculer de maniere recursive le nieme terme
    de la suite Un telle que :
    U0 = 2
    Un = 1/2 * ( Un-1 + 3 / Un-1 )'''
    if n == 0 : return (2)  # cas de base
    else :
        return( 0.5 * (Un(n - 1) + 3 / Un(n - 1)) )
```

## Exercice 13

```
# Conversion entier -> binaire

def conversion_it(n) :
    '''renvoie la conversion de l'entier n en binaire sour la forme d'une
    chaine de caracteres.'''
    res = ''    # initialisation de la chaine de caractere resultat
    while n != 0 :
        res = str(n % 2) + res # On concatene le reste trouve a gauche
        n = n // 2
    return(res)

def conversion_rec(n) :
    '''renvoie la conversion de l'entier n en binaire sour la forme d'une
    chaine de caracteres selon un algorithme recursif.'''
    if n == 1 or n == 0 :
        # cas de base
        return(str(n))
    else :
        return(conversion_rec(n // 2) + str(n % 2))
```

## Exercice 6 récursif

```
def terme_rec(a, n):
    '''version recursive'''
    if n == 0: # cas de base
        return a
    else:
        return pow(n + terme_rec(a, n - 1), 0.5)

def terme_rec_term(a, n, k = 1):
    '''version recursive terminale utilisant un argument par defaut'''
    if n == 0: # cas de base
        return a
    else:
        return terme_rec_term(pow(k + a, 0.5), n - 1, k + 1)

def terme_rec_term_bis(a, n):
    '''version recursive terminale utilisant une fonction auxiliaire'''
    def aux(acc, k):
        if k > n:
            return acc
        else:
            return aux(pow(k + acc, 0.5), k + 1)
    return aux(a, 1)
```

## Exercice 7 récursif

```
def produit_rec(a, b):
    '''version recursive'''
    if b == a:
        return b
    else:
        return a * produit_rec(a + 1, b)
```

## Exercice 8 récursif

Voici cinq solutions différentes !

— La première est la plus simple à comprendre, mais aussi la plus complexe en temps et en mémoire. En effet dans l'instruction `return(occ_rec(liste[ :-1], elt)` se cache une copie des $n-1$ termes de la liste, et donc une complexité cachée.
Cette subtilité n'est sans doute pas sanctionnée dans le cadre du programme d'IPT car la notion de fonction récursive est seulement présentée.
Le renvoi récursif ne comporte pas d'opérations, la récursivité est donc terminale.

— La seconde solution est nettement plus satisfaisante à ce niveau, même si, et c'est un peu paradoxal, elle est plus compliquée à comprendre ou à écrire. Elle utilise une fonction auxiliaire au sein de la fonction, et c'est cette fonction qui est à proprement parlée récursive. Cette façon d'écrire les fonctions récursives est classique avec *OCaml* le langage utilisé en option informatique en MP. Elle est également récursive terminale.
La version 2bis simplifie l'écriture des arguments de la fonction auxiliaire puisque `liste`, `elt` et `n` sont des variables définies dans l'espace local de la fonction `occ_rec2()` et sont donc définies dans celui de la fonction auxiliaire. C'est pour cela qu'il est inutile de les préciser dans les arguments de cette fonction.

— La troisième est basée sur le même principe que la seconde mais utilise une astuce de Python qui sait évaluer `a or b = True` dès que la proposition `a` est vrai sans avoir à évaluer la proposition `b`. Et cette opération s'arrêtera dès que le renvoi sera vrai. C'est donc encore une récursivité terminale.
— La quatrième est une exploitation des arguments à valeur par défaut que permet Python pour créer une fonction récursive terminale sans avoir besoin d'une fonction auxiliaire. Par contre, une complexité cachée est présente dans cette fonction. En effet, à chaque appel récursif, la fonction évalue la taille de la liste, ce qui est une opération de complexité linéaire en la taille de cette liste.
— La cinquième et dernière corrige le défaut de complexité de la quatrième en utilisant là encore une syntaxe Python qui permet de tester une instruction et d'en gérer les erreurs sous la forme d'exceptions (c'est une syntaxe hors programme).

```
def occ_rec1(liste, elt):
    if len(liste) == 0:
        return False
    elif liste[-1] == elt:
        return True
    else :
        return(occ_rec(liste[:-1], elt))

def occ_rec2(liste, elt):
    n = len(liste)
    def aux(i):
        if i == n:
            return False
        elif liste[i] == elt :
            return True
        else:
            return aux(i + 1)
    return aux(0)

def occ_rec3(liste, elt):
    n = len(liste)
    def aux(i):
        if i == n:
            return False
        else:
            return liste[i] == elt or aux(i + 1)
    return aux(0)

def occ_rec4(liste, elt, i = 0):
    if i > len(liste) - 1:
        return False
    else:
        return liste[i] == elt or occ_rec4(liste, elt, i + 1)

def occ_rec5(liste, elt, i = 0):
    try :
        e = liste[i]
    except IndexError:
        return False
    return e == elt or occ_rec4(liste, elt, i + 1)
```

## Exercice 9 récursif

```
def approx_rec(x, n):
    '''renvoie une approximation de la serie entiere Somme{x^n}
    pour x compris entre 0 et 1 non compris'''
    if n == 0:
        return 1
    else :
        return pow(x, n) + approx_rec(x, n - 1)
```

## Exercice 10 récursif

Là encore, je vous propose trois solutions diférentes pour cette fonction.

```
from math import factorial

def f_exp3(x, n):
    '''version recursive utilisant factorial() et donc trop complexe'''
    if n == 0 :
        return 1
    else :
        return pow(x, n) / factorial(n) + f_exp3(x, n - 1)

def f_exp4(x, n, facn):
    '''version recursive avec en troisieme argument la valeur de n!
    Il faut donc appeler la fonction ainsi : f_exp4(x, n, fact(n))'''
    if n == 0 :
        return 1
    else :
        return pow(x, n) / facn + f_exp4(x, n - 1, facn / n)

def f_exp5(x, n):
    '''version recursive beaucoup plus satisfaisante que f_exp4'''
    if n == 0:
        return 1
    else :
        def aux(i, fac):
            if i == n :
                return pow(x, i) / fac
            else :
                fac *= i
                return pow(x, i) / fac + aux(i + 1, fac)
        return 1 + aux(1, 1)
```