Cours 2 : Rappels et compléments sur les fonctions

Objectifs

- bonnes pratiques de codage,
- preuves de programme,
- récursivité.

Exercice 1 :
On donne la programmation de la fonction suivante.

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

- Que fait cette fonction ?
- Est-ce qu'elle est fiable ?
- Comment la modifier pour améliore votre temps de lecture ?

Introduction Dans l'industrie ou la recherche, un logiciel peut représenter des codes de millions de lignes avec une responsabilité humaine vitale comme dans les transports ou l'énergie. Il est donc essentiel de produire des programmes sûrs qui sont conformes aux exigences du cahier des charges.

La communauté informatique a mis en place des propositions d'amélioration du codage en Python, les PEP pour *Python Enhancement Proposal*. En voici quelques unes :

- *Beautiful is better than ugly*
- *Explicit is better than implicit*
- *Simple is better than complex*
- *Complex is better than complicated*
- *Flat is better than nested*
- *Sparse is better than dense*
- *Readability counts...*

Dans le cadre de cet enseignement, seule la programmation fonctionnelle est abordée.

1 Règles de conception des programmes

1. *L'analyse descendante* : parce qu'un algorithme simple diminue le risque d'erreurs, on procède par raffinements successifs.
2. *La modularité* : l'analyse descendante précédente aboutit à la décomposition du programme en plusieurs modules qui contiennent chacun une série de fonctions. Quand plusieurs modules sont regroupés ensemble, on parle de bibliothèque (*library* en anglais). Chacune de ces fonctions puis de ces modules peut être testée séparément. Une conséquence de cette décomposition en modules est la transmission des données sous forme d'*arguments* pour les fonctions. On privilégiera donc les *variables locales* aux variables globales.
3. *Les spécifications et les preuves de programme* : ils précisent le cahier des charges des éléments du programme et prouve que le programme se termine et qu'il aboutit au résultat recherché.
4. *La portabilité* : pour améliorer la portabilité de ses programmes aux différents systèmes d'exploitation, machines ou versions de logiciels, il est prudent de ne pas abuser des particularités d'un langage.
5. *La lisibilité* : insérer des commentaires au sein de son code et documenter systématiquement ses fonctions.

2 Spécifications et preuves de programme

L'objectif de cette section est de répondre à deux questions lors de la mise au point d'un algorithme : la *terminaison* et la *correction*. Et pour atteindre ces objectifs, il est également nécessaire de spécifier les conditions d'exécution de l'algorithme.

Remarque 1. *Pour la terminaison, le problème ne se pose que pour les boucles conditionnelles de type* `while`.

2.1 Spécifications

Définition 1. *Une* condition *pour un fragment de programme est une propriété portant sur les variables intervenant dans le fragment de programme.*

Définition 2. *Une* précondition *est une condition supposée vérifiée au moment où débute l'exécution du fragment de programme. Elle répond à la question : « Á quelles conditions entre-t-on dans l'algorithme ? ». Elle concerne donc les* arguments *de la fonction.*

Définition 3. *Une* postcondition *est une condition qui doit être vérifiée à la fin de l'exécution du fragment de programme. Elle répond à la question : « Á quelles conditions sort-on de l'algorithme ? ».*

Exercice 2 :
Spécifier la fonction de l'exemple 1.

Remarque 2. *Au moment de préciser les conditions liées aux arguments d'une fonction, on peut aussi ajouter qu'il est possible avec Python de déclarer des arguments avec* des valeurs par défaut. *La syntaxe est alors la suivante.*
`def fct(arg1, ..., argn, argpardef1 = val1, argpardef2 = val2, ...).`
Les arguments par défaut doivent être placés après les arguments obligatoires.

2.2 Terminaison d'une boucle

Pour démontrer que la boucle conditionnelle de cette fonction se termine, on va trouver un *variant de boucle*.

Définition 4. *Un* variant de boucle *est une quantité c qui vérifie les deux conditions suivantes :*
- $c \in \mathbb{N}$,
- *c décroit strictement à chaque itération. Comme il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il ne peut y avoir qu'un nombre fini d'itérations.*

Exercice 3 :
Montrer que les boucles conditionnelles de l'exemple 1 et des deux exemples ci-dessous se terminent en déterminant pour chacun de ces exemples un *variant de boucle*.

Exemple 2.
```
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)
```

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
```

Exercice 4 :
En mathématiques, on appelle *suite de Syracuse* (du nom d'une université américaine où le problème a été posé pour la première fois), ou *suite de Collatz* (Lothar Collatz, mathématicien allemand du XXe siècle qui énonça la conjecture associée) une suite d'entiers naturels définie de la manière suivante.

On part d'un nombre entier plus grand que zéro ; s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1. En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur. On s'arrête quand on arrive au chiffre 1.

Si on arrive à la valeur 1, la suite se répète indéfiniment selon un cycle donné appelé *cycle trivial*.

— Écrire une fonction affichant les termes de la suite de Syracuse d'un entier N donné.
— De même que pour les exemples précédents, montrer que votre programme se termine.

2.3 Conformité ou correction d'une boucle

Pour démontrer qu'un algorithme est correct, c'est à dire qu'il calcule bien ce que l'on attend, on établit un *invariant de boucle*.

Définition 5. *Un* invariant de boucle *est une propriété qui :*
— *est vérifiée avant d'entrer dans la boucle ;*
— *si elle est vérifiée avant une itération, alors elle est vérifiée après celle-ci ;*
— *permet d'en déduire que le programme est correct.*

Exercice 5 :
Montrer que les algorithmes des deux premiers exemples sont corrects.

2.4 Conclusion

En pratique, exhiber les conditions, un variant de boucle et un invariant sera suffisant pour prouver la terminaison et la correction d'une boucle. Et pour vos propres algorithmes, n'oubliez pas que *Explicit is better than implicit.*

Enfin, ce problème de terminaison peut devenir un problème mathématique complexe. L'objectif pour nous est de s'assurer que nos propres algorithmes se terminent et aboutissent au résultat recherché. Ces spécifications et preuves de programme doivent touver naturellement leurs places dans l'écriture de vos programmes, en utilisant la documentation de fonction et les commentaires.

2.5 Pour s'entraîner

Exercice 6 :
Écrire une fonction qui renvoie le n^{e} terme de la suite (U_n) définie par la relation de récurrence $U_n = \sqrt{n + U_{n-1}}$.

Exercice 7 :
Écrire une fonction qui renvoie le produit des entiers compris entre les entiers a et b.

Exercice 8 :
Écrire une fonction qui teste l'occurrence d'un élément dans une séquence.

Exercice 9 :
Écrire une fonction qui renvoie une approximation de la série entière suivante pour $x \in [0,1[$:

$$\frac{1}{1-x} = \sum_{n=0}^{+\infty} x^n$$

Exercice 10 :
Écrire une fonction qui renvoie une approximation de e^x par sa série entière :

$$e^x = \sum_{n=0}^{+\infty} \frac{x^n}{n!}$$

3 Récursivité

3.1 Récursivité simple

Définition 6 (Une mauvaise définition de ce type de fonction)**.** *Une* fonction récursive *est une fonction récursive.*

Définition 7 (Une bonne définition de ce type de fonction)**.** *On appelle* fonction récursive *une fonction qui comporte un appel à elle-même.*

Plus précisément, [1] page 33, une fonction récursive doit respecter ce qu'on pourrait appeler les « trois lois de la récursivité » :

1. une fonction récursive contient un cas de base ;
2. une fonction récursive doit modifier son état pour se ramener au cas de base ;
3. une fonction récursive doit s'apppeler elle-même.

Une fonction mathématique définie par une relation de récurrence (et une condition initiale, bien entendu), peut de façon naturelle être programmée de manière récursive.

Exemple 4. *Pour un réel x fixé, x^n peut se définir, comme fonction de n, par récurrence à partir des relations :*

$$x_0 = 1 \text{ et } x^n = x \cdot x^{n-1} \text{ si } n > 1$$

Le programme en Python s'écrit :

```
def puissance(x, n) :
    '''renvoie x^n'''
    if n > 0 :
        return ( x * puissance(x, n - 1) )
    else :
        return (1)
```

Modifions légèrement ce programme, pour bien en comprendre l'exécution :

```
def puissance(x, n) :
    '''renvoie x^n'''
    if n == 0 :
        return (1)
    else :
        print( '--' * n + '> appel de puissance ({}, {})'.format(x, n - 1) )
        y = x * puissance(x, n - 1)
        print( '--' * n + '> sortie de puissance ({}, {})'.format(x, n - 1) )
        return ( y )

puissance(2, 5)
```

```
----------> appel de puissance (2, 4)\\
--------> appel de puissance (2, 3)\\
------> appel de puissance (2, 2)\\
----> appel de puissance (2, 1)\\
--> appel de puissance (2, 0)\\
--> sortie de puissance (2, 0)\\
----> sortie de puissance (2, 1)\\
------> sortie de puissance (2, 2)\\
--------> sortie de puissance (2, 3)\\
----------> sortie de puissance (2, 4)
```

La machine applique la règle : `puissance(x, n) = x * puissance(x, n - 1)` *tant que l'exposant est différent de 0, ce qui introduit des calculs intermédiaires jusqu'à aboutir au cas de base :* `puissance(x, 0) = 1`*. Les calculs en suspens sont alors achevés dans l'ordre inverse jusqu'à obtenir le résultat final.*

3.2 Récursivité terminale

La récursivité terminale est un cas particulier de récursivité. C'est une fonction où l'appel récursif est la dernière instruction à être évaluée. Cette instruction est alors nécessairement 'pure', c'est à dire qu'elle consiste en un simple appel à la fonction, et jamais à un calcul ou une composition. Elle économise ainsi l'espace de la pile d'exécution, de ce point de vue, elle est donc comparable à un algorithme itératif.

Exercice 11 :
Programmer une fonction récursive terminale qui renvoie la somme des n premiers entiers.

3.3 Note sur les preuves de programme pour un algorithme récursif

La *terminaison* d'une fonction récursive se prouve par la décroissance stricte d'une certaine quantité entière, exemples :

— longueur d'une liste ;
— hauteur ou taille d'un arbre ;
— valeur d'un entier naturel.

La *correction* d'une fonction récursive se prouve par récurrence forte.

3.4 Pour s'entraîner

Exercice 12 (Calcul des termes d'une suite définie par récurrence, [2] page 130) **:**
Programmer un algorithme qui permette de calculer les termes de la suite (u_n) définie ci-dessous.

$$\begin{cases} u_0 = 2 \\ u_n = \frac{1}{2} \cdot \left(u_{n-1} + \frac{3}{u_{n-1}} \right) \end{cases}$$

Exercice 13 (Conversion entier → binaire) **:**
Écrire une fonction réalisant cette conversion en utilisant une fonction itérative, puis en utilisant une fonction récursive. La fonction renverra le résultat sous la forme d'une chaîne de caractères.

Reprendre les exercices 6 à 10 et proposer un algorithme récursif pour chacune des fonctions demandées.

4 Exercices à préparer pour le TD2

4.1 Encore un effort sur les structures de données

Exercice 14 :

```
def racines_trinome(a, b, c) :
    d = b ** 2 - 4 * a * c
    if d < 0 : return( complex(-b, - pow(-d, 0.5)) / (2 * a), \
                       complex(-b, pow(-d, 0.5)) / (2 * a) )
    if d == 0 : return(-b / (2 * a))
    if d > 0 : return((-b - pow(d, 0.5)) / (2 * a), (-b + pow(d, 0.5)) / (2 * a))
```

Cet algorithme n'est pas fonctionnel. En préciser la raison et proposer une correction.

Exercice 15 :
Petits casse-tête.

```
a, b, x = 4, 5, [1, 2]

def f(x):
    a, b = 1, 12
    x = x + 2
    def g(x):
        a, b = 3, 'a'
        print(a, b, x)
    g(x + 3)
    print(a, b, x)

f(x[b - a])
print(a, b, x)


L = [ [1, 2], [3, 4] ]
for val in L: val[0], val[1] = L[1][0], val[0]
print(L)
```

Préciser ce que renvoient ces deux programmes.

Exercice 16 :
Un programme à tester

```
def ma_fonction(n):
    nombre = ""
    q, r = n // 2, n % 2
    while (q, r) != (0, 0) :
        nombre = str(r) + nombre
        q, r = q // 2, q % 2
    return(int(nombre))

def mon_autre_fonction(n):
    if n == 1: return(1)
    else:      return(int(str(mon_autre_fonction(n // 2)) + str(n % 2)))

print(ma_fonction(14))
print(mon_autre_fonction(14))
```

Indiquer ce que renvoie ce programme. Est-il sûr que ces programmes renvoient le résultat attendu ?

4.2 Spécifications et preuves de programmes

Exercice 17 :
Prouver la terminaison de la boucle suivante en donnant un variant de boucle.

```
# n est un entier naturel
while n > 1 :
    if n % 2 == 0 :      n = n // 2
    else :               n = n + 1
```

4.3 Quatre algorithmes à connaître

Exercice 18 :
Écrire une version itérative, puis une version récursive pour calculer la factorielle d'un entier.

Montrer que votre algorithme termine et est correct.

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ (n-1)! \times n & \text{si } n \geqslant 1. \end{cases}$$

Exercice 19 :
Écrire une version itérative, puis une version récursive pour déterminer la racine d'une équation $f(x) = 0$ par la méthode de *dichotomie* (voir cours IPT de 1re année).

Exercice 20 (Exponentiation rapide) **:**
Pour élever un nombre à la puissance n, il existe un algorithme bien plus performant que la méthode naïve consistant à multiplier ce nombre $(n-1)$ fois par lui-même : il s'agit de la méthode dite d'*exponentiation rapide*. Étant donnés un réel positif a et un entier n, on remarque que :

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair}, \\ a \cdot \left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair}. \end{cases}$$

Cet algorithme se programme naturellement par une fonction récursive. Programmer cette exponentiation rapide et comparer sa performance à celle de la méthode itérative classique.

Exercice 21 (Algorithme de Ruffini-Horner) **:**
La méthode de Ruffini-Horner de recherche d'une valeur approchée de racine d'un polynôme est publiée à quelques années d'intervalle par Paolo Ruffini (1765-1822, italien) et par William George Horner (1786-1837, britannique) mais il semble bien que Horner n'ait pas eu connaissance des travaux de Ruffini. La méthode de Horner est ensuite popularisée par les mathématiciens De Morgan et J.R. Young.

Supposons que l'on souhaite calculer la valeur en x_0 du polynôme :

$$p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_n \cdot x^n$$

La première idée pour évaluer p en x_0 est de calculer chaque puissance de x_0 de manière naïve, de multiplier par les coefficients, puis de tout additionner.

Q 1. *Programmer la fonction naïve* `polynome()` *dont vous préciserez les arguments permettant de renvoyer la valeur* $p(x_0)$. *Pour cette fonction, je vous demande d'expliciter les spécifications : pré-conditions, post-condition, terminaison et correction.*

Q 2. *Évaluer le nombre de multiplications effectuées par cette fonction selon n, le degré du polynôme en supposant que ses coefficients* $a_i \neq 0,\ \forall i \in [\![0,n]\!]$.

On peut certes diminuer le nombre de multiplications à effectuer en utilisant l'algorithme d'exponentiation rapide.

La méthode de Horner permet de réduire encore le nombre de multiplications, en remarquant que :

$$p(x_0) = ((\ldots((a_n \cdot x_0 + a_{n-1}) \cdot x_0 + a_{n-2}) \cdot x_0 + \ldots) \cdot x_0 + a_1) \cdot x_0 + a_0$$

Ce faisant, il n'y a plus que n multiplications à effectuer !

Pour programmer cet algorithme, il suffit de calculer les n valeurs de la suite b_n définie par :

$$\begin{cases} b_n = a_n \\ \forall k \in [\![0, n-1]\!],\ b_k = a_k + b_{k+1} \cdot x_0 \end{cases}$$

Q 3. *Programmer l'algorithme de Horner de manière récursive (le polynôme p étant représenté par la liste de ses coefficients* $[a_0, \ldots, a_n]$*). Vous préciserez clairement le cas de base.*

5 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 5. *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 6. *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 3. *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 7.
```
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 : return 1 # cas de base
    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)
```

Références

[1] Alexandre Casamayou-Boucau et Pascal Chauvin et Guillaume Connan. *Programmation en Python pour les Mathématiques*. Dunod, collection Sciences Sup, 2012.

[2] Benjamin Wack et al. *Informatique pour tous en classes préparatoires aux grandes écoles*. Eyrolles, 2013.