TD n°1 et 2 : EXERCICES DE RÉVISION

Les exercices ci-dessous n'ont aucune prétention ; il s'agit seulement de révisions du programme de 1re année sur les instructions de base du langage Python.

I. Exercice 1

Écrire une fonction `binom(n,k)` permettant de calculer le coefficient binomial $\binom{n}{k}$ avec $n \in \mathbb{N}$ et $k \in \mathbb{Z}$ en utilisant exclusivement les propriétés suivantes :

1. Si $k < 0$ ou $k > n$, alors $\binom{n}{k} = 0$.
2. $\forall n \in \mathbb{N}$, $\binom{n}{0} = 1$.
3. $\forall k, n \in (\mathbb{N}^*)^2$, $\binom{n}{k} = \dfrac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$.

Attention : votre fonction doit renvoyer la valeur exacte du coefficient binomial, c'est-à-dire un entier avec tous ses chiffres, et non un nombre flottant approché.

Corrigé :

```python
def binom(n, k):
    # gestion simplifiée d'erreurs éventuelles:
    if not isinstance(n, int) or not isinstance(k, int):
        return "Erreur: n et k doivent être entiers"
    if n < 0:
        return "Erreur: n doit être positif."
    if k < 0 or k > n:
        return 0
    prod = 1
    for i in range(0, k):
        prod = ( prod * (n-i) ) // (i+1)
        # prod est forcément divisible par i+1 d'où  l'utilisation de la division entière
    return(prod)

if __name__ == "__main__":
    # la partie qui suit n'est exécutée que lorsqu'il s'agit du programme principal
    # et non lorsque ce programme est appelé depuis un autre via un import
    print(binom(50, 5)*binom(11,2))
    # nombre de combinaisons à l'Euro Millions
    print(binom(49,5)* binom(10,1))
    # nombre de combinaisons au Loto
```

```
116531800
19068840
```

II. Exercice 2

1. On souhaite construire le triangle de Pascal qui peut s'écrire de la façon suivante

$$\begin{array}{cccc} \binom{0}{0} & & & \\ \binom{1}{0} & \binom{1}{1} & & \\ \binom{2}{0} & \binom{2}{1} & \binom{2}{2} & \\ \binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} \end{array} \quad \text{ou encore} \quad \begin{array}{cccc} 1 & & & \\ 1 & 1 & & \\ 1 & 2 & 1 & \\ 1 & 3 & 3 & 1 \end{array}$$

Écrire une procédure `pascal(n)` qui construit ce triangle (sous forme d'une liste de listes de longueurs variables) en utilisant la fonction de l'exercice 1.
L'exemple précédent est donc le résultat de `pascal(3)`, qui devrait renvoyer la liste `[[1],[1,1],[1,2,1],[1,3,3,1]]`.

2. Il y a néanmoins une manière plus facile de calculer les termes successifs du triangle de Pascal en utilisant la formule éponyme :

$$\binom{n}{p} = \binom{n}{p-1} + \binom{n-1}{p-1}.$$

En utilisant la formule précédente, implémentez une fonction `pascal2(n)` qui renvoie le triangle de Pascal sans utiliser d'appel à la fonction `binom(n,p)`. Comparez la vitesse d'exécution des deux fontions `pascal(n)` et `pascal2(n)` pour $n = 50, 100, 200, 300, 400$. Représentez le résultat obtenu sous forme d'un graphique.

3. À présent que l'on sait construire les éléments du triangle de Pascal facilement, on peut résoudre le problème 203 du projet Euler. Les 8 premières lignes ($n = 7$) du triangle de Pascal s'écrivent

$$\begin{array}{ccccccccc}
1 & & & & & & & & \\
1 & 1 & & & & & & & \\
1 & 2 & 1 & & & & & & \\
1 & 3 & 3 & 1 & & & & & \\
1 & 4 & 6 & 4 & 1 & & & & \\
1 & 5 & 10 & 10 & 5 & 1 & & & \\
1 & 6 & 15 & 20 & 15 & 6 & 1 & & \\
1 & 7 & 21 & 35 & 35 & 21 & 7 & 1
\end{array}$$

It can be seen that the first eight rows of Pascal's triangle contain twelve distinct numbers : 1, 2, 3, 4, 5, 6, 7, 10, 15, 20, 21 and 35.
A positive integer n is called squarefree if no square of a prime divides n. Of the twelve distinct numbers in the first eight rows of Pascal's triangle, all except 4 and 20 are squarefree. The sum of the distinct squarefree numbers in the first eight rows is 105.
Find the sum of the distinct squarefree numbers in the first 51 rows of Pascal's triangle.

Plus précisément, écrivez un programme `squarefree_pascal(n)` qui calcule la somme des nombres « squarefree » distincts présents dans les n premières lignes du triangle de Pascal.

Corrigé :

```python
from copy import *
from math import sqrt
import time

def binom(n, k):
    # voir exo 1
    prod = 1
    for i in range(0, k):
        prod = ( prod * (n-i) ) // (i+1)
    return(prod)

def pascal1(N):
    '''Renvoie le triangle de Pascal sous forme de liste de listes en
    utilisant la définition des coefficients binomiaux.'''
    pascal = []
    for n in range(N+1):
        nouvelle_ligne = []
        for p in range(n+1):
            nouvelle_ligne.append(binom(n, p))
        pascal.append(nouvelle_ligne)
    return pascal

def pascal2(n):
    '''Renvoie le triangle de Pascal sous forme de liste de listes en
    utilisant la formule de Pascal pour calculer la ligne suivante.'''
    pascal= [ [1] ] # première ligne binom(0,0)
    for i in range(1, n+1):
        # On construit la ligne suivante en mettant d'abord des 1
        ligne_suivante = [1] * (i+1)
        # puis en utilisant la formule bien connue pour les coefficients non extrêmes
        for p in range(1, i):
            ligne_suivante[p] = pascal[i-1][p] + pascal[i-1][p-1]
        pascal.append(ligne_suivante)
    return pascal

def is_squarefree(n):
    '''Détermine si un entier n est 'squarefree' ou non.'''
```

```python
    if n == 0:
        return False
    # On teste tous les carrés à partir de deux et jusqu'à la racine de n
    for i in range(2,int(sqrt(n))+2): # +1 à cause d'éventuelles erreurs d'arrondi
        if n % (i*i) == 0:
            return False
    return True

def somme_squarefree_pascal(n):
    '''Détermine la somme des entiers 'squarefree' distincts du triangle de Pascal.'''
    triangle = pascal2(n)
    sqrfree = []
    # on stocke dans sqrfree les éléments distincts et 'squarefree'
    # on aurait pu utiliser la structure d'ensemble (set) mais est-elle au programme?
    for L in triangle:
        for x in L:
            if x not in sqrfree and is_squarefree(x):
                sqrfree.append(x)
    return sum(sqrfree)

print(pascal1(6))
print(pascal2(6))
#comparaison des temps
n = 500
t1 = time.time()
pascal1(n)
t2 = time.time()
pascal2(n)
t3 = time.time()
print()
print( " {} secondes avec pascal1 pour {} lignes \n".format(t2-t1,n))
print( " {} secondes avec pascal2 pour {} lignes \n".format(t3-t2,n))
print(somme_squarefree_pascal(6))
print(somme_squarefree_pascal(51))
```

```
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1], [1, 5, 10, 10, 5, 1], [1, 6, 15, 20, 15, 6, 1]]
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1], [1, 5, 10, 10, 5, 1], [1, 6, 15, 20, 15, 6, 1]]

 4.002003192901611 secondes avec pascal1 pour 500 lignes

 0.03125286102294922 secondes avec pascal2 pour 500 lignes

42
34029210557389
```

III. Exercice 3

Considérons le triangle suivant :

3
7 4
2 **4** 6
8 5 **9** 3

En partant du sommet et en descendant uniquement selon les chiffres adjacents sur la ligne du dessous, la somme maximale que l'on peut obtenir est $3+7+4+9=23$. Écrivez une fonction **`somme_maximale(triangle)`** qui, étant donné un triangle proposé comme une liste de listes, calcule la somme maximale sur un des chemins qui mène du sommet à la base.

Une petite mise en garde néanmoins : pour un triangle de N lignes, il existe 2^N-1 chemins différents du sommet à la base. S'il est plausible de tous les explorer par une méthode « force brute » pour de faibles valeurs de N (disons jusqu'à 20 environ), cela n'est plus possible pour un triangle de 100 lignes. La méthode se doit d'être un peu plus réfléchie.

Corrigé :

On va faire correspondre au triangle précédent un triangle « dual » qui contient à chaque emplacement la somme du plus grand des trajets permettant d'atteindre ce point. On construit alors la ligne suivante en ajoutant au point courant la plus grande valeur du chemin depuis les deux points parents qui peuvent mener au point courant. Pour l'exemple de l'énoncé, cela donnerait le triangle dual :

```
3
10 7
12 14 13
20 19 23 16
```

dont il suffit de prendre la valeur maximale de la dernière ligne pour répondre à la question posée.

```python
def somme_maximale(triangle):
    '''Associe à un triangle la somme maximale des nombres selon un chemin qui
    va de haut en bas. L'idée est de  calculer une ligne après l'autre
    du triangle "dual" qui contient le maximum de la somme sur le chemin qui
    mène là où on regarde.'''
    dual = [triangle[0][:]]      # Initialisation
    for i in range(1,len(triangle)):
        ligne = triangle[i]
        largeur = len(ligne)
        # Rajout d'une nouvelle ligne
        nouvelle = [0]*largeur
        # Cas particulier du début et de la fin de la ligne
        # qui n'ont qu'un seul  parent possible
        nouvelle[0] = ligne[0] + dual[-1][0]
        nouvelle[largeur - 1] = ligne[largeur - 1] + dual[-1][largeur - 2]
        for j in range(1,largeur-1):
        #  on prend le maximum des deux parents
            nouvelle[j] = ligne[j] + max(dual[-1][j-1],dual[-1][j])
        dual.append(nouvelle)
    # Il reste à renvoyer le maximum de la dernière ligne et le tour est joué !
    return max(dual[-1])

triangle=[ [3], [7, 4], [2, 4, 6], [8, 5, 9, 3] ]
print(somme_maximale(triangle))
```

```
23
```

IV. Exercice 4

Soit f continue strictement monotone sur un intervalle $[a,b]$, telle que $f(a)f(b)<0$. On sait alors que f s'annule une fois et une seule sur $]a,b[$.

Écrire une fonction **`dicho(f,a,b,eps)`** permettant d'encadrer le zéro de f à **`eps`** près.

Pour tester, on pourra chercher le point fixe de la fonction cos sur $\left[0\,;\frac{\pi}{2}\right]$, en considérant la fonction $f : x \mapsto x - \cos x$.

Comparer le résultat obtenu avec celui que l'on obtient en considérant la suite définie par récurrence par $u_{n+1} = f(u_n)$.

Corrigé :

```python
from math import *

def f(x):
    return cos(x)

def g(x):
    return x - f(x)

def dicho(f, a, b, eps):
    #on suppose a<b et f(a)*f(b)<0
    n = 0
    while b - a > eps:
        c = (a + b) / 2
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
        n += 1
    return c, n

def point_fixe(f, a, b, eps):
    u0 = (a + b) / 2
    u1 = f(u0)
    n = 0
```

```
25      while abs(u1 - u0) > eps:
26          u0 = u1
27          u1 = f(u0)
28          # ou directement u0, u1 = u1, f(u0)
29          n += 1
30      return (u0 + u1)/2, n
31
32  eps = 1e-10
33  a, b = 0, 1.5
34
35  c,n = dicho(g, a, b, eps)
36  print(" Valeur par dichotomie: {} en {} étapes".format(c,n))
37
38  c,n = point_fixe(f, a, b, eps)
39  print(" Valeur par suite récurrente: {} en {} étapes".format(c,n))
```

```
 Valeur par dichotomie: 0.7390851332747843 en 34 étapes
 Valeur par suite récurrente: 0.7390851332081732 en 49 étapes
```

V. Exercice 5

1. Écrire une fonction qui renvoie, sous forme de liste, les chiffres d'un entier naturel n (on n'utilisera pas les fonctions sur les chaînes de caractères).
2. Écrire une fonction qui renvoie la somme des carrés des chiffres d'un entier n.
3. Un entier est un *nombre heureux* si, lorsque l'on calcule la somme des carrés de ses chiffres puis la somme des carrés des chiffres du nombre obtenu et ainsi de suite, on aboutit au nombre 1.
   Écrire une fonction qui teste si un nombre est heureux. On utilisera pour cela le fait que, si l'on applique un tel processus à partir d'un entier quelconque, on finit toujours par obtenir soit $\{1\}$, soit un nombre de l'ensemble $\{4, 16, 37, 58, 89, 145, 42, 20\}$ (qui est alors malheureux).
4. Afficher la liste de tous les nombres heureux inférieurs à 100.
5. Afficher la liste de tous les couples heureux $(n, n+1)$ et de tous les triplets heureux $(n, n+, n+2)$ avec $n \leqslant 10000$. Généraliser.

Pour tester vos résultats, on donne :

- la liste des nombres heureux entre 1 et 100 :

$$\{1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97\}$$

- les premiers couples heureux :

$$(31, 32),\ (129, 130),\ (192, 193),\ (262, 263), \ldots$$

- les premiers triplets heureux :

$$(1880, 1881, 1882),\ (4780, 4781, 4782),\ (4870, 4871, 4872), \ldots$$

- les premiers quadruplets heureux :

$$(7839, 7840, 7841, 7842),\ (8769, 8740, 8741, 8742),\ (11248, 11249, 11250, 11251), \ldots$$

Corrigé :

```
1   def chiffres_avec_Python(n):
2       # on triche en utilisant la conversion nombre--> chaîne de caractères
3       return list(map(int, list(str(n))))
4
5   def chiffres(n):
6       # liste des chiffres, à l'envers
7       L = []
8       while n > 9:
9           L.append(n % 10)
10          n = n // 10
11      L.append(n)
12      return L
13
14  def somme_des_carres_des_chiffres(n):
15      return sum([i*i for i in chiffres(n)])
16
17  def est_heureux(n):
```

```python
    fini = False
    while not fini:
        if n == 1:
            heureux = True
            fini = True
        if n in {0, 4, 16, 37, 58, 89, 145, 42, 20}:
            heureux = False
            fini = True
        n = somme_des_carres_des_chiffres(n)
    return heureux

def liste_des_heureux(n):
    L = []
    for i in range(1, n+1):
        if est_heureux(i):
            L.append(i)
    # ou plus pythonnesque:
    # return [i for i in range(1,n+1) if est_heureux(i)]
    return L

def liste_des_suites_joyeuses(n,p):
    # p-uplets heureux
    L=[]
    for i in range(1, n+1):
        k = 0
        while est_heureux(i+k):
            k += 1
        convient = (k == p)
        if convient:
            L.append([i+k for k in range(0, p)])
    return(L)

def liste_des_suites_joyeuses2(n,p):
    #autre version plus rapide suite à une suggestion
    # d'une élève
    L = liste_des_heureux(n)
    Result = []
    for i in range(0, len(L) - p):
        L1 = L[i:i+p]
        if L1[-1] - L1[0] == p - 1:
            Result.append(L1)
    return Result

print(est_heureux(7), est_heureux(50),'\n')
print(liste_des_heureux(100),'\n')
print(liste_des_suites_joyeuses2(300,2),'\n')
print(liste_des_suites_joyeuses2(5000,3),'\n')
print(liste_des_suites_joyeuses2(20000,4),'\n')
```

```
True False

[1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100]

[[31, 32], [129, 130], [192, 193], [262, 263]]

[[1880, 1881, 1882], [4780, 4781, 4782], [4870, 4871, 4872]]

[[7839, 7840, 7841, 7842], [8739, 8740, 8741, 8742], [11248, 11249, 11250, 11251], [12148, 12149, 12150, 12151]]
```

VI. Exercice 6

Un nombre entier est dit *tricolore* si son carré s'écrit uniquement avec les chiffres 1, 4 et 9.

1. Écrire une fonction `tricolore(n)` permettant de vérifier si un entier n est tricolore.
2. Écrire une fonction `liste_tricolores(N)` qui donne la liste de tous les entiers tricolores $\leqslant N$.

Corrigé :

```python
def chiffres(n):
    # on utilise les fonctions Python, plus rapides
    return list(map(int, list(str(n))))

def is_tricolore(n):
    x = n * n
    L = chiffres(x)
    tricolore = True
    i = 1
    while i < len(L) and tricolore:
        if not L[i] in [1, 4, 9]:
            tricolore = False
        i += 1
    return tricolore

def liste_tricolores(N, M):
    # liste des nombres tricolores entre N et M
    L = []
    for i in range(N, M+1):
        if is_tricolore(i):
            L.append(i)
    # plus pythonnesque:
    # return [i for i in range(N, M+1) if is_tricolore(i)]
    return(L)

print(liste_tricolores(10,1000000))
```

```
[12, 21, 29, 38, 107, 212, 771, 2538, 24788, 31488, 70107, 72107, 264479, 387288, 590857, 717021]
```

VII. Exercice 7

Écrire un programme qui permet d'afficher la liste des nombres premiers inférieurs à un entier N donné, en utilisant la méthode du crible d'Ératosthène.

Cette méthode consiste à considérer la table formée des entiers de 2 à N ; dans cette table, on raye déjà les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant.

On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant puisque tout nombre non premier admet forcément un diviseur inférieur ou égal à sa racine.

Corrigé :

```python
from time import *

def crible1(N):
    # Première version, où pour rayer un élément, on le retire de la liste
    liste = list(range(2, N+1))
    racine = int( N ** 0.5) + 1 # +1 pour éventuelles erreurs d'arrondi
    for i in liste:
        if i <= racine:
            for j in liste:
                if (j > i) and (j % i == 0):
                    liste.remove(j)
        else:
            break
    return liste

# pour vérifier éventuellement
# print(crible1(1000))

debut = time()
N = 50000
crible1(N)
print("Temps 1ère méthode: ", time() - debut, "pour N = ",N)

# Le gros défaut de cette méthode est le temps prohibitif des opérations
# liste.remove() qui obligent à chaque fois à recréer une nouvelle liste

# on utilisera donc une liste fixe où liste[i]= True ou False selon que i est premier ou non

```

```python
def crible2(N):
    liste = [True] * (N + 1)
    liste[0] = False
    liste[1] = False
    # on raye déjà les nombres pairs
    for k in range(2, N // 2 + 1):
        liste[2 * k] = False
    racine = int( N ** 0.5) + 1
    for i in range(3, racine + 1, 2):
        if liste[i]:
            for k in range(i, N // i + 1):
                liste[i * k] = False
    prem = [i for i in range(0, N+1) if liste[i]]
    return prem

# pour vérifier
# print(crible2(1000))

debut = time()
N = 10000000 # 10^7
crible2(N)
print("Temps 2ème méthode", time() - debut, "pour N = ",N)

def crible3(N):
    # la même chose mais en utilisant les fonctions Python sur les listes
    liste = [True] * (N + 1)
    liste[0] = False
    liste[1] = False
    # on raye déjà les nombres pairs
    liste[4 : : 2] = [False] * (N//2 - 1)
    racine = int( N ** 0.5) + 1
    for i in range(3, racine + 1, 2):
        if liste[i]:
            liste[i*i : : i] = [False] *  (N//i -i + 1)
    prem = [i for i in range(0, N+1) if liste[i]]
    return prem

# pour vérifier
# print(crible3(10000))

debut = time()
N = 10000000 # 10^7
crible3(N)
print("Temps 2ème méthode + listes Python", time() - debut, "pour N = ",N)
```

```
Temps 1ère méthode:  4.064594745635986 pour N =  50000
Temps 2ème méthode 2.003147840499878 pour N =  10000000
Temps 2ème méthode + listes Python 0.6126677989959717 pour N =  10000000
```

VIII. Exercice 8

Soit f la fonction définie par :

$$f : n \longmapsto \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n+1 & \text{si } n \text{ est impair} \end{cases}$$

On s'intéresse à la suite définie par récurrence par :

$$u_0 \in \mathbb{N} \quad \text{et} \quad u_{n+1} = f(u_n).$$

Par exemple, avec $u_0 = 13$, on obtient

$$13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$$

C'est ce qu'on appelle la suite de Collatz (ou suite de Syracuse) du nombre 13. Après avoir atteint le nombre 1, la suite de valeurs (1,4,2,1,4,2,1,4,2...) se répète indéfiniment en un cycle de longueur 3 appelé cycle trivial.

La conjecture de Collatz est l'hypothèse selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1. Bien qu'elle ait été vérifiée pour les 5,7 premiers milliards de milliards d'entiers et en dépit de la simplicité de son énoncé, cette conjecture défie depuis de nombreuses années les mathématiciens. Paul Erdös a dit à son propos que « les mathématiques ne sont pas encore prêtes pour de tels problèmes... »

1. Implémenter la fonction `f(n)` ci-dessus.

2. Soit $u_0 = a \in \mathbb{N}^*$. On appelle orbite de a la liste des termes de la suite $u_{n+1} = f(u_n)$ jusqu'à ce que l'on tombe sur 1 (compris). Écrire le programme `orbite(a)` qui prend comme entrée l'entier `a` et qui renvoie son orbite sous forme d'une liste.

Plusieurs caractéristiques d'une orbite peuvent être explorées :

- son « temps de vol » correspond au nombre total d'entiers visités sur l'orbite.
- son « altitude » est donnée par le plus grand entier visité sur l'orbite.
- son « temps de vol en altitude » correspond au nombre d'étapes avant de passer strictement en dessous du nombre de départ.
- et enfin son « temps de vol avant la chute » correspond au nombre d'étapes minimum après lequel on ne repasse plus au-dessus de la valeur de départ.

Par exemple pour l'orbite du nombre 13, l'altitude vaut 40 (maximum de la suite), le temps de vol vaut 10, le temps de vol en altitude vaut 3 (on passe à $10 < 13$ lors de la 3^e étape) et le temps de vol avant la chute vaut 6 (on passe à $8 < 13$ lors de la 6^e itération de la fonction `f`).

3. Écrire une fonction `temps_de_vol(a)` qui renvoie le temps de vol correspondant à l'orbite de `a`.

4. Écrire une fonction `altitude(a)` qui renvoie l'altitude de l'orbite de `a`. Pour s'entraîner, on implémentera la recherche du maximum « à la main ».

5. Écrire une fonction `temps_en_altitude(a)` qui renvoie le temps de vol en altitude correspondant à l'orbite de `a`.

6. Écrire une fonction `temps_avant_chute(a)` qui renvoie le temps de vol avant la chute pour l'orbite de `a`.

Nous allons utiliser ces procédures pour résoudre les questions suivantes.

7. Pour des entiers de départ de valeur strictement inférieures à un million, déterminer à chaque fois

a) celui qui monte le plus haut en altitude et la valeur de cette altitude maximale (à stocker dans la liste `le_plus_haut` qui aura donc deux éléments `[a,altitude(a)]`).

b) celui dont le temps de vol est le plus long et la valeur de ce temps de vol (à stocker dans la liste `le_plus_long`).

c) celui dont le temps de vol en altitude est le plus long et la valeur de ce temps de vol (à stocker dans la liste `le_plus_long_en_altitude`).

d) celui dont le temps de vol avant la chute est le plus long et la valeur de ce temps de vol (à stocker dans la liste `le_plus_long_avant_la_chute`).

8. Ne trouvez-vous pas que la procédure suggérée fasse un peu « gâchis » ? Implémentez une procédure unifiée qui puisse effectuer tous les calculs précédents en moins de temps. Estimez le gain de temps que l'on peut espérer et mesurez-le (via un `import time` et à l'aide de `time.time()`, plus d'info avec `help(time.time)`). N'oubliez pas de tester votre procédure en retrouvant les résultats précédent

---

Corrigé :

```python
import time

def f(n):
    """ Fonction pour définir la suite de Syracuse """
    if n%2 == 0:
        return n//2
    else:
        return 3*n+1

def orbite(a):
    """ Renvoie la liste des termes de la suite u_{n+1}=f(u_n) jusqu'à ce que
    l'on obtienne 1 (compris)."""
    L=[]
    x = a
    while x != 1:
        L.append(x)
        x = f(x)
    L.append(1)
    return L

def temps_de_vol(a):
    """ Renvoie le temps de vol correspondant à l'orbite de a. """
    return len(orbite(a))

def altitude(a):
    """ Renvoie l'altitude de l'orbite de a (implémentée à la main). """
```

```
    # Il s'agit de chercher le maximum de la liste
    # sans utiliser de fonction Python toute prête
    max = 0
    for n in orbite(a):
        if n > max:
            max = n
    return max

def temps_en_altitude(a):
    """ Renvoie le temps de vol en altitude correspondant à l'orbite de a."""
    # Il s'agit en fait de déterminer la position du premier élément inférieur à a dans la liste.
    # On pourrait bien sûr utiliser la fonction orbite, mais ce n'est pas la peine:
    # on calcule le début de l'orbite et on s'arrête dès qu'un élément de la suite est < a
    if a == 1:
        return 1 # car dans ce cas la boucle suivante ne termine pas!
    temps = 0
    x = a
    while x >= a:
        x = f(x)
        temps += 1
    return temps

def temps_avant_chute(a):
    """Renvoie le temps de vol avant chute correspondant à l'orbite de a."""
    # On part de la fin  et on s'arrête dès qu'on dépasse a
    L = orbite(a)
    i = len(L) - 1
    while L[i] < a:
         i -= 1
    return i + 1

def temps_avant_chute2(a, L):
    # idem mais ici on passe la valeur de l'orbite à la fonction
    i = len(L) - 1
    while L[i] < a:
         i -= 1
    return i + 1

def cherche_max(fonction, nmax = 10**6):
    # On calcule bestialement toutes les valeurs
    L = [fonction(a) for a in range(1,nmax)]
    # on utilise la fonction max de Python pour gagner un peu de temps
    maximum = max(L)
    # si il y a plusieurs indices correspondant au maximum
    # il faudrait écrire:
    #      indices_max = [i+1 for i, j in enumerate(L) if j == maximum]
    # si le max n'est atteint qu'une seule fois on peut écrire
    indice_max = L.index(maximum)+1
    return [indice_max, maximum]

def big_liste_orbite(nmax = 10**6):
    # On construit la liste de toutes les orbites en réutilisant
    # celles déjà calculées
    # et on calcule toutes les données demandées au fur et à mesure
    max_altitude = 0
    max_temps_de_vol = 0
    max_temps_avant_chute = 0
    max_temps_en_altitude = 0
    big_liste = [ [0], [1] ]
    for a in range(2,nmax):
        # détermination de l'orbite de a
        L = []
        i = 0  # désigne le nombre de termes de l'orbite qui sont >= a
        x = a
        while x >= a:
            # calcul des termes successifs de l'orbite de a
            # on en profite pour mettre à jour l'altitude maximum
            if x > max_altitude:
                max_altitude = x
                imax_altitude = a
            L.append(x)
            x = f(x)
            i +=1
        # on ajoute l'orbite déjà calculée dès que l'on a atteint une valeur < a
        L.extend(big_liste[x])
        # et on stocke l'orbite complète dans bigliste
        big_liste.append(L)
```

```
        #
        if i > max_temps_en_altitude:
            max_temps_en_altitude = i
            imax_temps_en_altitude = a
        #
        longueur = len(L)
        if longueur > max_temps_de_vol:
            max_temps_de_vol = longueur
            imax_temps_de_vol = a
        #
        t = temps_avant_chute2(a,L)
        if t > max_temps_avant_chute:
            max_temps_avant_chute = t
            imax_temps_avant_chute = a

    return([imax_altitude,max_altitude],[imax_temps_de_vol, max_temps_de_vol], \
            [imax_temps_en_altitude, max_temps_en_altitude],\
            [imax_temps_avant_chute,max_temps_avant_chute])


t1 = time.time()
print(cherche_max(altitude))           # -> [704511, 56991483520]
print(cherche_max(temps_de_vol))       # -> [837799, 525]
print(cherche_max(temps_en_altitude))  # -> [626331, 287]
print(cherche_max(temps_avant_chute))  # -> [886953, 357]
t2 = time.time()
print("temps 1ere méthode: ",t2-t1)

t1 = time.time()
L = big_liste_orbite()
t2 = time.time()
print(L)
print("temps 2eme méthode: ", t2-t1)
```

```
[704511, 56991483520]
[837799, 525]
[626331, 287]
[886953, 357]
temps 1ere méthode:  79.75080466270447
([704511, 56991483520], [837799, 525], [626331, 287], [886953, 357])
temps 2eme méthode:  12.156582593917847
```

IX. Exercice 9

Cet exercice concerne le procédé d'orthonormalisation de Schmidt, dont je rappelle brièvement le principe.

Dans un espace préhilbertien réel, il s'agit, à partir d'une famille libre de vecteurs $(v_1, \dots, v_n)$, de construire une famille orthogonale $(e_1, \dots, e_n)$ qui engendre les mêmes sous-espaces vectoriels successifs, c'est-à-dire telle que la matrice de passage d'une base à l'autre soit triangulaire supérieure.

L'étape générale de l'algorithme consiste à soustraire au vecteur v_{k+1} sa projection orthogonale sur le sous-espace vectoriel $\text{Vect}(v_1, \dots, v_k) = \text{Vect}(e_1, \dots, e_k)$, puis à le normer. Il peut donc s'écrire :

$$\left|\begin{array}{ll} \textbf{Initialisation :} & \text{On pose } u_1 = v_1 \text{ puis } e_1 = \dfrac{u_1}{\|u_1\|} \\ \textbf{k-ième étape :} & \text{On pose } u_k = v_k - \displaystyle\sum_{j=1}^{k-1} \langle e_j \,|\, v_k \rangle \, e_j \text{ puis } e_k = \dfrac{u_k}{\|u_k\|} \cdot \end{array}\right.$$

A. Orthonormalisation de vecteurs dans $\mathbb{R}^m$.

1. Implémenter ce procéder en créant une fonction **gramschmidt** qui prend en entrée une liste de n vecteurs de $\mathbb{R}^m$, chaque vecteur étant lui-même une liste de longueur m, puis qui renvoie la famille orthonormale obtenue. On veillera à afficher un message d'erreur si la famille n'est pas libre ; cela correspond, dans l'algorithme ci-dessus, au cas où l'un des u_k est le vecteur nul, c'est-à-dire, en Python, si $\|u_k\| < 10^{-10}$ (par exemple).

2. La méthode précédente est instable numériquement à cause des erreurs d'arrondi qui font que, dans certains cas, les vecteurs u_j ne sont pas vraiment orthogonaux. Pour l'illustrer, considérer la famille de vecteurs de $\mathbb{R}^4$:

$$v_1 = (1, \varepsilon, 0, 0) \quad , \quad v_2 = (1, 0, \varepsilon, 0) \quad , \quad v_2 = (1, 0, 0, \varepsilon)$$

avec $\varepsilon = 10^{-10}$ (par exemple). Que constatez-vous ?

L'algorithme de Gram-Schmidt modifié consiste à calculer chaque u_k pour $k \geqslant 2$ à l'aide des formules suivantes (je vous laisse le soin de vérifier que, mathématiquement, cela donne la même chose) :

$$\begin{array}{rcl} u_k^{(0)} & = & v_k \\ u_k^{(1)} & = & v_k - p_{u_1}(v_k)\,, \\ u_k^{(2)} & = & u_k^{(1)} - p_{u_2}(u_k^{(1)})\,, \\ & \vdots & \\ u_k^{(k-2)} & = & u_k^{(k-3)} - p_{u_{k-2}}(u_k^{(k-3)}) \\ u_k & = & u_k^{(k-2)} - p_{u_{k-1}}(u_k^{(k-2)}) \end{array}$$

où p_u désigne le projeté orthogonal sur la droite de base u, c'est-à-dire $p_u(v) = \dfrac{\langle u \,|\, v\rangle}{\|u\|^2}\, u$.

Implémenter cet algorithme et comparer avec le résultat précédent.

Corrigé :

```python
from math import sqrt

def add_vecteurs(l1, l2):
    # somme de deux vecteurs
    return [x + y for x, y in zip(l1, l2)]
    # ou plus simplement:
    # return [[l1[i] + l2[i} for i in range(len(l1))]

def mult_ext(lambd, v):
    # multiplication du vecteur v par le réel lambd, lambda = mot réservé!
    return [lambd*x for x in v]

def prod_scal(l1, l2):
    # produit scalaire usuel dans R^n
    # on suppose que les listes l1 et l2 ont même longueur
    return sum(x * y for x, y in zip(l1, l2))
    # ou plus simplement:
    # return sum([l1[i]*l2[i} for i in range(len(l1))])

def norme(l):
    return sqrt(prod_scal(l, l))

def aff_mat(l):
    # affichage matrice avec arrondis 4 décimales
    for i in range(len(l)):
        print('[',end=' ')
        for j in range(len(l[0])):
            print( "{:.4f}".format(l[i][j]), end=' ')
        print(']')
    print()

def verifortho(l):
    # vérifie si la famille est bien orthonormale en calculant sa matrice de Gram
    # on doit obtenir la matrice identité
    for i in range(len(l)):
        for j in range(len(l)):
            print( "{:.4f}".format(prod_scal(ortho[i], ortho[j])), end=' ')
        print()
    print()

def gramschmidt(listev):
    eps = 1e-12
    n = len(listev) #nombre de vecteurs
    m = len(listev[0]) # dimension de l'ev
    # on suppose que la liste contient au moins un vecteur et qu'il n'est pas nul!
    norm = norme(listev[0])
    ortho=[ mult_ext(1/norm, listev[0]) ] # liste des vecteurs de la bon finale
    for k in range(1, n):
        projete = [0]*m # projeté orthogonal
        for i in range(k):
            ps = prod_scal(ortho[i], listev[k])
            projete = add_vecteurs(projete, mult_ext(-ps, ortho[i]))
        u = add_vecteurs(listev[k], projete)
        norm = norme(u)
        if norm < eps:
```

```
                return "Erreur, famille liée"
            ortho.append(mult_ext(1/norm,u))
        return(ortho)

    def gramschmidt_modif(listev):
        eps = 1e-12
        n = len(listev) #nombre de vecteurs
        m = len(listev[0]) # dimension de l'ev
        # on suppose que la liste contient au moins un vecteur et qu'il n'est pas nul!
        ortho = []
        for k in range(n):
            u = listev[k]
            for i in range(k):
                ps = prod_scal(ortho[i], u)
                projete = mult_ext(-ps, ortho[i])
                u = add_vecteurs(u, projete)
            norm = norme(u)
            if norm < eps:
                return "Erreur, famille liée"
            ortho.append(mult_ext(1/norm,u))
        return(ortho)

    l = [ [1,2,1], [2,-1,1], [3,2,2] ]
    ortho = gramschmidt(l)
    aff_mat(ortho)
    verifortho(ortho)

    a = 1e-10
    l = [ [1,a,0,0], [1,0,a,0], [1,0,0,a] ]
    ortho = gramschmidt(l)
    aff_mat(ortho)
    verifortho(ortho)

    l = [ [1,2,1], [2,-1,1], [3,2,2] ]
    ortho = gramschmidt_modif(l)
    aff_mat(ortho)
    verifortho(ortho)

    a = 1e-10
    l = [ [1,a,0,0], [1,0,a,0], [1,0,0,a] ]
    ortho = gramschmidt_modif(l)
    aff_mat(ortho)
    verifortho(ortho)
```

```
[ 0.4082 0.8165 0.4082 ]
[ 0.7591 -0.5521 0.3450 ]
[ 0.5071 0.1690 -0.8452 ]

1.0000 -0.0000 -0.0000
-0.0000 1.0000 0.0000
-0.0000 0.0000 1.0000

[ 1.0000 0.0000 0.0000 0.0000 ]
[ 0.0000 -0.7071 0.7071 0.0000 ]
[ 0.0000 -0.7071 0.0000 0.7071 ]

1.0000 -0.0000 -0.0000
-0.0000 1.0000 0.5000
-0.0000 0.5000 1.0000

[ 0.4082 0.8165 0.4082 ]
[ 0.7591 -0.5521 0.3450 ]
[ 0.5071 0.1690 -0.8452 ]

1.0000 -0.0000 -0.0000
-0.0000 1.0000 0.0000
-0.0000 0.0000 1.0000

[ 1.0000 0.0000 0.0000 0.0000 ]
[ 0.0000 -0.7071 0.7071 0.0000 ]
[ 0.0000 -0.4082 -0.4082 0.8165 ]

1.0000 -0.0000 -0.0000
-0.0000 1.0000 -0.0000
-0.0000 -0.0000 1.0000
```

B. Orthonormalisation de polynômes.

Un polynôme $\sum_{k=1}^{n} a_k X^k$ sera représenté en Python par une liste de longueur $n+1$ contenant ses coefficients.

Le but de l'exercice est d'écrire un programme qui orthonormalise par le procédé de Schmidt la famille de polynômes $(1, X, X^2, \dots, X^n)$ (n entier quelconque) lorsqu'on munit $\mathbb{R}[X]$ du produit scalaire

$$\langle P \,|\, Q \rangle = \int_{-1}^{1} P(t)Q(t)\,\mathrm{d}t.$$

Pour cela on sera amené à écrire quelques procédures utiles pour travailler sur les polynômes. J'en cite quelques-unes, mais la liste n'est pas exhaustive :

- une procédure **`valeur(P, x)`** qui calcule la valeur d'un polynôme P en x, par l'algorithme de Hörner ;
- une procédure **`mult_ext(c, P)`** qui multiplie un polynôme P par une constante c ;
- une procédure **`add_poly(P, Q)`** qui additionne deux polynômes P et Q ;
- une procédure **`mult_poly(P, Q)`** qui multiplie deux polynômes P et Q ;
- une procédure **`primitive(P)`** qui renvoie une primitive du polynôme P ;
- une procédure **`affiche_poly(P)`** pour ceux qui veulent faire un joli affichage....
- une procédure **`prod_scal(P, Q)`** pour calculer le produit scalire de P et Q ;
- etc..

Remarque : il existe un package **`numpy.polynomial`** *qui contient ce genre de procédures, mais nous ne l'utiliseraons pas ici, le but de ce TD étant de vous faire programmer !*

Corrigé :

```python
from math import sqrt

def valeur(p,x):
    """
    Evalue le polynôme p en x. On utilise le schéma de Horner
    """
    n = len(p) - 1
    val = p[n]
    for i in range(n-1,-1,-1):
        val = val*x + p[i]
    return val

def coef(p,k):
    """ Renvoie le coefficient de x^k dans p """
    if k < len(p):
        return p[k]
    else:
        return 0

def add_poly(p,q):
    """ Renvoie la somme de 2 polynômes."""
    return [coef(p,k) + coef(q,k) for k in range(max(len(p),len(q)))]

def mult_const(p,c):
    """ Renvoie p multiplié par une  constante"""
    return [c * p[i] for i in range(len(p))]

def mult_poly(p,q):
    """ Renvoie le produit de 2 polynômes"""
    return [ sum([ coef(p,k) * coef(q, n-k) for k in range(n+1) ])\
                  for n in range(len(p)+len(q)-1) ]
def primitive(p):
    """Renvoie la primitive de p avec la constante 0 """
    return [0]+[coef(p,k)/(k+1) for k in range(len(p))]

def affiche_poly(p):
    """
    pour un joli affichage du polynôme p
    enfin, faut pas être trop difficile...
    """
    def affiche_monome(n):
        c = coef(p,n)
        c = int(c*10000)/10000 # 4 chiffres après la virgule
        if c == 0:
            return ''
```

```
            signe = ' + ' if c >= 0 else  ' - '
            if n > 1:
                monome = 'X^%d' % n
            elif n == 1:
                monome = 'X'
            else:
                monome=''
            if abs(c) == 1 and n!=0:
                return signe + monome
            else:
                return signe + str(abs(c)) +( chr(183) if n!=0 else '') + monome

        s=''
        for n in range(len(p),-1,-1):
           s = s + affiche_monome(n)

        # on enleve le premier signe si c'est un +
        if len(s) >2 and s[0:3] == ' + ':
            s = s[3:]
        return s

    def produit_scalaire(p,q):
        """ le produit scalaire : intégrale de -1 à 1 du produit p*q """
        res = primitive(mult_poly(p,q))
        return valeur(res,1) - valeur(res,-1)

    def monome(d):
        """ le polynome X^d """
        return d*[0]+[1]

    def Schmidt(a):
        """
        orthonormalisation par le procédé de Schmidt
        de la famille de polynomes stockée dans la liste a
        """
        n = len(a)
        norme = sqrt(produit_scalaire(a[0], a[0]))
        ortho = [mult_const(a[0], 1/norme)]
        for k in range(1, n):
            projete = [0]
            for i in range(k):
                ps = produit_scalaire(ortho[i], a[k])
                projete = add_poly(projete, mult_const(ortho[i], -ps))
            b = add_poly(a[k], projete)
            norme = sqrt(produit_scalaire(b,b))
            ortho.append(mult_const(b,1/norme))
        return ortho

    # programme principal
    base_canonique = [monome(d) for d in range(6)]
    bon = Schmidt(base_canonique)
    for i in range(len(bon)):
        print(affiche_poly(bon[i]))
```

```
0.7071
1.2247·X
2.3717·X^2 - 0.7905
4.677·X^3 - 2.8062·X
9.2807·X^4 - 7.9549·X^2 + 0.7954
18.4685·X^5 - 20.5205·X^3 + 4.3972·X
```

C. Application à la recherche d'un polynôme annulateur d'une matrice.

Soit A une matrice de $\mathcal{M}_n(\mathbb{R})$. Puisque $\mathcal{M}_n(\mathbb{R})$ est un espace vectoriel de dimension n^2, la famille des puissances de A, $\left\{A^k,\ 0 \leqslant k \leqslant n^2\right\}$ est liée puisqu'elle comporte n^2+1 vecteurs. Il existe donc une relation de dépendance linéaire entre ces puissances, c'est-à-dire un polynôme P tel que $P(A)=0$. P s'appelle un polynôme annulateur de A.

L'espace vectoriel $\mathcal{M}_n(\mathbb{R})$ sera muni du produit scalaire canonique donné par

$$\langle A \,|\, B\rangle = \operatorname{tr}({}^tAB) = \sum_{i,j} a_{ij}b_{ij}.$$

Si l'on applique l'algorithme de Gram-Schmidt à la famille formée des puissances successives de A, cette famille étant liée, il existe nécessairement un rang k où, en reprenant les notations du début de l'exercice, le vecteur u_k est nul, c'est-à-dire où le vecteur v_k est combinaison linéaire des précédents. Le but de l'exercice est d'utiliser cette idée afin de trouver un polynôme annulateur d'une matrice A donnée, c'est à dire un plus petit entier k tel que A^k s'exprime comme combinaison linéaire des A^j pour $0 \leqslant j \leqslant k-1$.

Comme exemple, vous pourrez considérer la matrice $A = \begin{pmatrix} 3 & -4 & 0 & 2 \\ 4 & -5 & -2 & 4 \\ 0 & 0 & 3 & -2 \\ 0 & 0 & 2 & -1 \end{pmatrix}$ et vérifier que $A^4 = 2A^2 - I_4$.

Corrigé :

```python
from math import sqrt

# procédures sur les polynômes

def coef(p, k):
    """ Renvoie le coefficient de x^k dans p """
    if k < len(p):
        return p[k]
    else:
        return 0

def add_poly(p, q):
    """ Renvoie la somme de 2 polynômes."""
    return [coef(p,k) + coef(q,k) for k in range(max(len(p),len(q)))]

def mult_poly_const(c, p):
    """ Renvoie p multiplié par une  constante"""
    return [c * p[i] for i in range(len(p))]

def affiche_poly(p):
    """
    pour un joli affichage du polynôme p
    enfin, faut pas être trop difficile...
    """
    def affiche_monome(n):
        c = coef(p,n)
        c = int(c*10000)/10000 # 4 chiffres après la virgule
        if c == 0:
            return ''
        signe = ' + ' if c >= 0 else  ' - '
        if n > 1:
            monome = 'X^%d' % n
        elif n == 1:
            monome = 'X'
        else:
            monome=''
        if abs(c) == 1 and n!=0:
            return signe + monome
        else:
            return signe + str(abs(c)) +( chr(183) if n!=0 else '') + monome

    s=''
    for n in range(len(p),-1,-1):
       s = s + affiche_monome(n)

    # on enleve le premier signe si c'est un +
    if len(s) >2 and s[0:3] == ' + ':
        s = s[3:]
    return s

# procédures sur les matrices

def add_mat(A, B):
    n = len(A)
    return [ [A[i][j]+B[i][j] for j in range(n)] for i in range(n) ]

def mult_mat_const(c, A):
    n = len(A)
    return [ [c*A[i][j] for j in range(n)] for i in range(n) ]

def mat_nulle(n):
    return [ [  0 for _ in range(n) ] for _ in range(n) ]

def mat_identite(n):
    A = mat_nulle(n)
    for i in range(len(A)):
        A[i][i] = 1
    return A
```

```python

def mult_mat(A, B):
    n = len(A)
    C = mat_nulle(n)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k]*B[k][j]
    return C

def affiche_mat(A):
    # affichage matrice avec arrondis 4 décimales
    for i in range(len(A)):
        print('[',end=' ')
        for j in range(len(A)):
            print( "{:.4f}".format(A[i][j]), end=' ')
        print(']')
    print()

def produit_scalaire(A, B):
    n = len(A)
    s = 0
    for i in range(n):
        for j in range(n):
            s += A[i][j]*B[i][j]
    return s

def norme2(A):
    return produit_scalaire(A, A)

def poly_annulateur(A):
    eps = 1e-10
    n = len(A)
    B = mat_identite(n) # B contiendra les puissances successives de A
    poly = [ [1] ]
    ortho = [ B ]
    for k in range(1, n+1):
        B = mult_mat(B, A)
        P = [0]
        projete = mat_nulle(n)
        for i in range(k):
            coeff = -produit_scalaire(ortho[i], B)/norme2(ortho[i])
            projete = add_mat(projete, mult_mat_const(coeff, ortho[i]))
            P = add_poly(P, mult_poly_const(coeff, poly[i]))
        C = add_mat(B, projete)
        ortho.append(C)
        P.append(1)
        poly.append(P)
        if norme2(C) < eps:
            return P
    return "erreur"

# programme principal

# exemple de la feuille
A = [ [3,-4,0,2], [4,-5,-2,4], [0,0,3,-2],[0,0,2,-1] ]
print("Un polynôme annulateur de la matrice: \n")
affiche_mat(A)
print("est: ",affiche_poly(poly_annulateur(A)))

# une matrice compagnon
A = [ [0,0,0,0,1], [1,0,0,0,2], [0,1,0,0,3], [0,0,1,0,4], [0,0,0,1,5] ]
print("Un polynôme annulateur de la matrice: \n")
affiche_mat(A)
print("est: ",affiche_poly(poly_annulateur(A)))
```

```
Un polynôme annulateur de la matrice:

[ 3.0000 -4.0000 0.0000 2.0000 ]
[ 4.0000 -5.0000 -2.0000 4.0000 ]
[ 0.0000 0.0000 3.0000 -2.0000 ]
[ 0.0000 0.0000 2.0000 -1.0000 ]

est:  X^4 - 2.0·X^2 + 1.0
Un polynôme annulateur de la matrice:

[ 0.0000 0.0000 0.0000 0.0000 1.0000 ]
[ 1.0000 0.0000 0.0000 0.0000 2.0000 ]
[ 0.0000 1.0000 0.0000 0.0000 3.0000 ]
[ 0.0000 0.0000 1.0000 0.0000 4.0000 ]
[ 0.0000 0.0000 0.0000 1.0000 5.0000 ]

est:  X^5 - 4.9999·X^4 - 4.0·X^3 - 3.0·X^2 - 2.0·X - 1.0
```