TD n°1 : 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. $\forall k, n \in (\mathbb{N}^*)^2$, $\binom{n}{k} = \dfrac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$.

Corrigé :

```python
def binom(n, k):
    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

Écrire une fonction qui, étant donnée une durée en secondes, affiche le temps correspondant en jours, heures, minutes et secondes.

Corrigé :

```python
def affiche(duree):
    # afffichage dune durée fournie en jours, minutes
    # heures et secondes, sous forme d'une liste ou d'une séquence
    j, h, m, s = duree
    duree_en_s = 86400*j + 3600*h + 60*m + s
```

```
    s = " Une durée de {} secondes correspond à: \n {} jours,\
{} heures, {} minutes et {} secondes.".format(duree_en_s,j,h,m,s)
    return s

def conversion(duree):
    # convertit une durée en secondes en jours, heures etc..
    minutes_en_s = 60
    heures_en_s = 60 * minutes_en_s
    jours_en_s = 24 * heures_en_s
    L = []
    for i in [jours_en_s, heures_en_s, minutes_en_s]:
        L.append(duree // i)
        duree = duree % i
    L.append(duree)
    return L

print(affiche(conversion(100000)))
```

```
 Une durée de 100000 secondes correspond à:
 1 jours,3 heures, 46 minutes et 40 secondes.
```

III. Exercice 3

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 :

```
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
```

IV. Exercice 4

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):
18      fini = False
19      while not fini:
20          if n == 1:
21              heureux = True
22              fini = True
23          if n in {0,4,16,37,58,89,145,42,20}:
24              heureux = False
25              fini = True
26          n = somme_des_carres_des_chiffres(n)
27      return heureux
28
29  def liste_des_heureux(n):
30      L = []
31      for i in range(1,n+1):
32          if est_heureux(i):
33              L.append(i)
34      # ou plus pythonnesque
35      # return [i for i in range(1,n+1) if est_heureux(i)]
36      return L
37
38  def liste_des_suites_joyeuses(n,p):
39      # p-uplets heureux
40      L=[]
41      for i in range(1,n+1):
42          k = 0
43          while est_heureux(i+k):
44              k += 1
45          convient = (k == p)
46          if convient:
47              L.append([i+k for k in range(0,p)])
48      return(L)
49
50  def liste_des_suites_joyeuses2(n,p):
51      #autre version plus rapide suite à une suggestion
52      # d'une élève
53      L = liste_des_heureux(n)
54      Result = []
55      for i in range(0, len(L) - p):
56          L1 = L[i:i+p]
57          if L1[-1] - L1[0] == p - 1:
58              Result.append(L1)
59      return Result
60
61  print(est_heureux(7), est_heureux(50),'\n')
62  print(liste_des_heureux(100),'\n')
63  print(liste_des_suites_joyeuses2(300,2),'\n')
64  print(liste_des_suites_joyeuses2(5000,3),'\n')
65  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], [121
```

V. Exercice 5

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 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 = 0
    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)
```

```
40
41 print(liste_tricolores(10,1000000))

   [12, 21, 38, 107, 212, 31488, 70107, 387288]
```

VI. Exercice 6

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 :

```
1  from time import *
2
3  def crible1(N):
4      # Première version, où pour rayer un élément,
5      # on le retire de la liste
6      liste = list(range(2, N+1))
7      racine = int( N ** 0.5) + 1
8      for i in liste:
9          if i <= racine:
10             for j in liste:
11                 if (j > i) and (j % i == 0):
12                     liste.remove(j)
13         else:
14             break
15     return liste
16
17 # pour vérifier éventuellement
18 # print(crible1(1000))
19
20 debut = time()
21 N = 50000
22 crible1(N)
23 print("Temps 1ère méthode: ", time() - debut, "pour N = ",N)
24
25 # Le gros défaut de cette méthode est le temps
26 # prohibitif des opérations liste.remove() qui obligent à chaque fois à
27 # recréer une nouvelle liste
28
29 # on utilisera donc une liste fixe où liste[i]= True ou False selon
30 # que i est premier ou non
31
32 def crible2(N):
33     liste = [True] * (N + 1)
34     liste[0] = False
35     liste[1] = False
36     # on raye déjà les nombres pairs
37     for k in range(2, N // 2 + 1):
38         liste[2 * k] = False
39     racine = int( N ** 0.5) + 1
40     for i in range(3, racine + 1, 2):
41         if liste[i]:
42             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 = 1000000 # 10^6
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:  5.6263978481292725 pour N =  50000
Temps 2ème méthode 0.18417620658874512 pour N =  1000000
Temps 2ème méthode + listes Python 0.740211009979248 pour N =  10000000
```

VII. Exercice 7

Un marchand de légumes très maniaque souhaite ranger ses petits pois en les regroupant en boîtes de telle sorte que chaque boîte contienne un nombre factoriel de petits pois.

Il souhaite également utiliser le plus petit nombre de boîtes possible. Ainsi, s'il a 17 petits pois, il utilisera :

- 2 boîtes de $3! = 6$ petits pois, soit 12 petits pois rangés ;
- 2 boîtes de $2! = 2$ petits pois, soit 4 petits pois rangés ;
- 1 boîte de $1! = 1$ petit pois, soit 1 petit pois rangé.

ce qui donne bien $2 \times 3! + 2 \times 2! + 1 \times 1! = 12 + 4 + 1 = 17$.

D'une manière générale, s'il a **nb_petits_pois** , il doit trouver une suite $a_1, a_2, \ldots, a_p$ d'entiers positifs ou nuls avec $a_p > 0$ et telle que

$$\text{nb\_petits\_pois} = a_1 \times 1! + a_2 \times 2! + \cdots + a_p \times p!$$

avec $a_1 + \cdots + a_p$ minimal.

Écrire un programme **range_petits_pois(nb_petits_pois)** qui renvoie la liste **[a1,a 2,..., ap]** (unique) qui permettra à l'assistant du marchand de préparer les boîtes avant d'y ranger ses petits pois.

Corrigé :

```python
def range_petits_pois(nb_petits_pois):
    # recherche du plus grand p ( + 1) tel que p! <= nb_petits_pois
    # au passage on stocke les factorielles pour ne pas les recalculer
    p = 1
    factorielle=1
    liste_fact = []
    while factorielle <= nb_petits_pois:
        liste_fact.append(factorielle)
        p += 1
        factorielle *= p
    liste_fact.reverse()
    p -= 1
    # on range
    reste = nb_petits_pois
    L = []
    for k in range(0,p):
        L.append(reste // liste_fact[k])
        reste %= liste_fact[k]
    L.reverse()
    return L

print (range_petits_pois(17))
```

```
[1, 2, 2]
```

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 3e étape) et le temps de vol avant la chute vaut 6 (on passe à $8 < 13$ lors de la 6e 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.clock()** , plus d'info avec **help(time.clock)**). N'oubliez pas de tester votre procédure en retrouvant les résultats précédent

Corrigé :

```
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 tombe sur 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
```

```
30      for n in orbite(a):
31          if n > max:
32              max = n
33      return max
34
35  def temps_en_altitude(a):
36      """ Renvoie le temps de vol en altitude correspondant à l'orbite de a."""
37      # Il s'agit en fait de déterminer la position du premier élément
38      # inférieur à a dans la liste.
39      if a == 1:   # dans ce cas la boucle ci-dessous ne marche pas
40          return 1
41      L = orbite(a)
42      i = 1
43      while L[i] >= a:
44          i += 1
45      return i
46
47  def temps_avant_chute(a):
48      """Renvoie le temps de vol avant chute correspondant à l'orbite de a."""
49      # On part de la fin  et on s'arrête dès qu'on dépasse a
50      L = orbite(a)
51      i = len(L) - 1
52      while L[i] < a:
53           i -= 1
54      return i + 1
55
56  def temps_avant_chute2(a,L):
57      # idem mais ici on passe l'orbite directement
58      i = len(L) - 1
59      while L[i] < a:
60           i -= 1
61      return i + 1
62
63  def cherche_max(fonction, nmax = 10**6):
64      # On calcule bestialement toutes les valeurs
65      L = [fonction(a) for a in range(1,nmax)]
66      # on utilise la fonction max de Python pour gagner un peu de temps
67      maximum = max(L)
68      # si il y a plusieurs indices correspondant au maximum
69      # il faudrait écrire:
70      #      indices_max = [i+1 for i, j in enumerate(L) if j == maximum]
71      # si le max n'est atteint qu'une seule fois on peut écrire
72      indice_max = L.index(maximum)+1
73      return [indice_max,maximum]
74
75  def big_liste_orbite(nmax = 10**6):
76      # On construit la liste de toutes les orbites en réutilisant
77      # celles déjà calculées
78      # et on calcule toutes les données demandées au fur et à mesure
79      max_altitude = 0
80      max_temps_de_vol = 0
81      max_temps_avant_chute = 0
82      max_temps_en_altitude = 0
83      big_liste = [ [0], [1] ]
84      for a in range(2,nmax):
85          L = []
86          i = 0
87          x = a
```

```
88          while x >= a:
89              if x > max_altitude:
90                  max_altitude = x
91                  imax_altitude = a
92              L.append(x)
93              x = f(x)
94              i +=1
95          # on ajoute l'orbite déjà calculée dès que l'on atteint
96          # une valeur < a
97          L.extend(big_liste[x])
98          big_liste.append(L)
99          #
100         if i > max_temps_en_altitude:
101             max_temps_en_altitude = i
102             imax_temps_en_altitude = a
103         #
104         longueur = len(L)
105         if longueur > max_temps_de_vol:
106             max_temps_de_vol = longueur
107             imax_temps_de_vol = a
108         #
109         t = temps_avant_chute2(a,L)
110         if t > max_temps_avant_chute:
111             max_temps_avant_chute = t
112             imax_temps_avant_chute = a
113
114     return([imax_altitude,max_altitude],[imax_temps_de_vol, max_temps_de_vol], \
115             [imax_temps_en_altitude, max_temps_en_altitude],\
116             [imax_temps_avant_chute,max_temps_avant_chute])
117
118
119 t1 = time.clock()
120 print(cherche_max(altitude))            # -> [704511, 56991483520]
121 print(cherche_max(temps_de_vol))        # -> [837799, 525]
122 print(cherche_max(temps_en_altitude))   # -> [626331, 287]
123 print(cherche_max(temps_avant_chute))   # -> [886953, 357]
124 t2 = time.clock()
125 print("temps 1ere méthode: ",t2-t1)
126
127 t1 = time.clock()
128 L = big_liste_orbite()
129 t2 = time.clock()
130 print(L)
131 print("temps 2eme méthode: ", t2-t1)
```

```
[704511, 56991483520]
[837799, 525]
[626331, 287]
[886953, 357]
temps 1ere méthode:  145.4038308873807
([704511, 56991483520], [837799, 525], [626331, 287], [886953, 357])
temps 2eme méthode:  14.049454860660433
```

IX. Exercice 9

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)

```

```python
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.'''
    ligne = [1]
    pascal= [[1]]
    for i in range(1,n+1):
        # On construit la ligne suivante
        ligne_suivante = [1]
        for p in range(1,len(ligne)):
            ligne_suivante.append(ligne[p]+ligne[p-1])
        ligne_suivante.append(1)
        ligne = copy(ligne_suivante)
        pascal.append(ligne_suivante)
    return pascal

def is_squarefree(n):
    '''Détermine si un entier n est 'squarefree' ou non.'''
    # 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):
        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)
    s = []
    # on stocke dans s les éléments distincts
    for L in triangle:
        for x in L:
            if x not in s:
                s.append(x)
    # On ne prend que les 'squarefree'
    sqrfree = [n for n in s if is_squarefree(n)]
    return sum(sqrfree)

print(pascal1(7))
#comparaison des temps
n = 500
t1 = time.clock()
pascal1(n)
t2 = time.clock()
pascal2(n)
t3 = time.clock()
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(7))
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
 4.558354830938839 secondes avec pascal1 pour 500 lignes

 0.020904709351611928 secondes avec pascal2 pour 500 lignes

105
34029210557389
```

X. Exercice 10

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 liste comme ceux de la section précédente, 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.

```
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 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
```

XI. Exercice 11 (E3A PSI 2015)

Le jeu d'échec se joue sur un échiquier, c'est à dire sur un plateau de 8×8 cases. Ces cases sont référencées de a1 à h8 (voir figures).

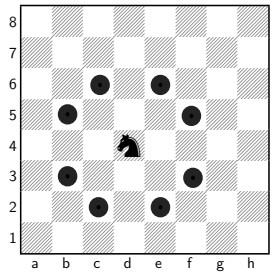

FIGURE 1
déplacements permis

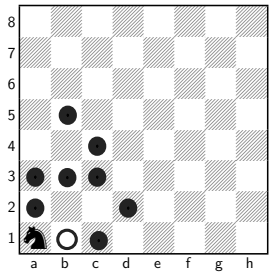

FIGURE2
un exemple

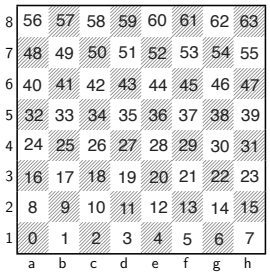

FIGURE 3
numérotation

Une pièce, appelée le cavalier, se déplace suivant un "L" imaginaire d'une longueur de deux cases et d'une largeur d'une case.
Exemple (figure 1) : un cavalier situé sur la case d4 atteint, en un seul déplacement, une des huit cases b5, c6, e6, f5, f3, e2, c2 ou b3.
Dans toute la suite de l'exercice, on appellera **case permise** toute case que le cavalier peut atteindre en un déplacement à partir de sa position.

> Le but de cet exercice est d'écrire un programme faisant parcourir l'ensemble de l'échiquier à un cavalier **en ne passant sur chaque case qu'une et une seule fois**

Motivation et méthode retenue

Une première idée est de faire parcourir toutes les cases possibles à un cavalier en listant à chaque déplacement les cases parcourues. Lorsque celui-ci ne peut plus avancer, on consulte le nombre de cases parcourues.

- Si ce nombre est égal à $64 = 8 \times 8$, alors le problème est résolu.
- Sinon, il faut revenir en arrière et tester d'autres chemins.

1. **Exemple** : on considère le parcours suivant d'un cavalier démarrant en a1 (figure 2)

   a1, b3, c1, a2, c3, b5, a3, c4, d2

   Avec ce début de parcours, au déplacement suivant :

   a) le cavalier va en b1. Peut-il accomplir sa mission ?

   b) le cavalier ne va pas en b1. Peut-il accomplir sa mission ?

Il convient donc dans la résolution du problème proposé de se retrouver dans la situation repérée en cette première question.

Dans tout ce qui suit, nous nommerons **coordonnées** d'une case la liste d'entiers $[i, j]$ où i représente le numéro de ligne et j le numéro de colonne (tous deux compris entre 0 et 7). Par exemple, la case b3 a pour coordonnées $[2, 1]$.
D'autre part, les cases sont numérotées de 0 à 63 en partant du coin gauche comme indiqué en figure 3.
Nous appellerons *indice* d'une case, l'entier $n \in [|0, 63|]$ ainsi déterminé. b3 a, par exemple, un indice égal à 17.

2. Écrire une fonction `indice` qui prend en argument la liste des coordonnées d'une case est renvoie son indice. Ainsi, `indice([2,1])` doit être égal à 17.

3. Écrire une fonction *coord* qui à l'indice n d'une case associe la liste $[i,j]$ de ses coordonnées. Ainsi `coord(17)` doit être égal à $[2,1]$.

4. On considère la fonction Python `CasA` suivante :

```
def CasA(n):
    Deplacements=[[1,-2],[2,-1],[2,1],[1,2],[-1,2],[-2,1],[-2,-1],[-1,-2]]
    L=[]
    i,j=Coord(n)
    for d in Deplacements:
        u=i+d[0]
        v=j+d[1]
        if u>=0 and u<8 and v>=0 and v<8:
            L.append(Indice([u,v]))
    return(L)
```

a) Que renvoient `CasA(0)` et `CasA(39)`.

b) Expliquer en une phrase ce que fait cette fonction.

5. Écrire une fonction `Init` ne prenant aucun argument et qui modifie deux variables globales `ListeCA` et `ListeCoups`. `ListeCoups` recevra la iste vide. `ListeCA` recevra une liste de 64 éléments. Chaque élément `listeCA[n]` (pour $0 \leqslant n \leqslant 63$) devra contenir la liste des indices des cases qu'un cavalier peut atteindre en un coup à partir de la cas d'indice n.

6. Après exécution de la fonction `Init()`, la commande `ListeCA[0]` renvoie-t-elle $[5]$, $[10,17]$, $[10,17,0]$, $[17,0,10]$, $[]$ ou une autre valeur ?

7. Au cours de la recherche, lorsqu'on déplace le cavalier vers la case d'indice n, cet indice n doit être retiré de la liste des *cases permises* à partir de la position n.
Exemple : après exécution de la fonction `Init()`, la liste des cases permises depuis b1 est $[a3, c3, d2]$ et `ListeCA[1]=[16,18,11]`.
La liste des cases permises depuis a3 est $[b5, c4, c2, b1]$ et `ListeCA[16]=[33,26,10,1]`.
Puis, on choisit de commencer le parcours en posant le cavalier en b1. Cette case doit donc être retirée de la liste des cases permises de a3, c3 et d2. En particulier pour a3, la liste `ListeCA[16]` devient `[33,26,10]`.
Cette méthode nous permet de détecter les blocages : le cavalier arrive sur la case d'indice n, n est alors retiré de toutes les listes `ListeCA[k]` pour toute case `k` permise pour n. Si dès lors l'une de ces listes devient vide, nous dirons que nous somme alors dans une *situation critique*, cela signifiera que la case d'indice `k` ne peut plus être atteinte que depuis la case d'indice n. Par conséquent,

- si le cavalier se déplace sur une autre case que celle d'indice `k`, alors cette dernière ne pourra plus jamais être atteinte ;
- si le cavalier se déplace sur la case d'indice `k`, il est bloqué pour le coup suivant. Soit la mission est accomplie, soit le cavalier n'a pas parcouru toutes les cases.

Le programme va réaliser la recherche en maintenant à jour la variable globale `ListeCoups` afin qu'elle contienne en permanence la liste des positions successives occupées par le cavalier au cours de ses tentatives de déplacement. Nous avons alors besoin d'écrire trois fonctions.

a) Écrire une fonction `OccupePosition` qui

- prend comme argument un entier n (indice d'une case), l'ajoute à la fin de la variable globale `ListeCoups`,
- puis enlève n de toutes les listes `ListeCA[k]` pour toutes les cases `k` permises depuis la case d'indice n,
- renvoie enfin la valeur `True` si nous sommes dans une situation critique et `False` sinon.

On pourra utiliser la méthode `remove` *qui permet de retirer d'une liste le premier élément égal à l'argument fourni. Si l'argument ne fait pas partie de la liste, une erreur sera retournée*

```
L=[1,2,3,4,5,6]
L.remove(2) # modifie L en [1,3,4,5,6]
L.remove(6) # provoque une erreur
```

b) Écrire une fonction `LiberePosition` qui ne prend pas d'argument et qui
- récupéré le dernier élément `n` de la variable globale `ListeCoups` (i.e. l'indice de la dernière case jouée à l'aide de la fonction `OccupePosition`),
- puis l'enlève de `ListeCoups`,
- et enfin, qui ajoute `n` à toutes les listes `ListeCA[k]` pour toutes les cases d'indice `k` permises depuis la case d'indice `n`.

On pourra utiliser la méthode `pop` *qui renvoie le dernier élément d'une liste et le supprime de cette même liste.*

```
L=[1,2,3,4,2,5,2]
n=L.pop()  #n=2 et L=[1,2,3,4,2,5]
```

c) Écrire une fonction `TestePosition` d'argument un entier `n` (indice d'une case) qui :
- occupe la position d'indice `n`,
- vérifie si la situation est critique.
  Si c'est le cas, la fonction vérifiera si les 63 cases sont occupées et, dans ce cas renverra `True` pour indiquer que la recherche est terminée. Si les 63 cases ne sont pas occupées, la fonction libèrera la case d'indice `n` et renverra `False`.
  Dans le cas contraire, la fonction vérifiera avec `TestePosition` toutes les cases d'indice `k` jouables après celle d'indice `n` (on prendra garde à affecter une variable locale avec la liste `ListeCA[n]` puisque celle-ci risque d'être modifiée lors des appels suivants). La fonction retournera `True` dès que l'un des appels à `TestePosition` retourne `True` ou libèrera la case d'indice `n` et retournera `False` sinon.

8. Afin de réduire notablement la complexité temporelle du programme, on part du principe qu'il faut tester en priorité les cases ayant le moins de cases permises possibles. On appellera *valuation* d'une case d'indice `n` le nombre de cases permises pour cette case.

a) Écrire une fonction `valuation` qui prend comme argument un indice `n` de case en entrée et renvoie la valuation de cette case.

b) Écrire une fonction `Fusion` qui prend comme arguments deux listes `A` et `B` d'entiers entre 0 et 63 ; on suppose ces listes triées par ordre croissant de valuation de leurs éléments ; l'appel `fusion(A,B)` retourne comme valeur la liste fusionnée de tous les éléments de `A` et `B` triée par ordre croissant de valuation de ses éléments.

c) Écrire une fonction `TriFusion` qui prend en argument une liste `L` d'entiers compris entre 0 et 63 et qui retourne comme valeur la liste de tous les éléments de `L` triée par valuation croissante de ses élèments.

d) Modifier la fonction `TestePosition` pour qu'elle agisse ainsi que l'on a décidé en début de question.

Corrigé :

```
from copy import *

Taille = 8 # Taille de l'échiquier
NbCases = Taille**2

def Indice(L):
    return Taille*L[0] + L[1]

def Coord(n):
    return [n//Taille, n%Taille]

def CasA(n):
    Deplacements=[[1,-2],[2,-1],[2,1],[1,2],[-1,2],[-2,1],[-2,-1],[-1,-2]]
    L=[]
    i,j = Coord(n)
    for d in Deplacements:
        u = i + d[0]
        v = j + d[1]
        if u>=0 and u<Taille and v>=0 and v<Taille:
```

```
                L.append(Indice([u,v]))
    return(L)

def Init():
    global ListeCA, ListeCoups
    ListeCoups=[]
    ListeCA=[CasA(n) for n in range(NbCases)]

def OccupePosition(n):
    global ListeCA, ListeCoups
    ListeCoups.append(n)
    critique = False
    for k in ListeCA[n]:
        ListeCA[k].remove(n)
        # pas utile de faire un test, ne provoquera pas d'erreur
        # car si k est dans ListeCA[n] alors n est dans ListeCA[k]!
        if len(ListeCA[k]) == 0:
            critique = True
    return critique

def LiberePosition():
    global ListeCA, ListeCoups
    n = ListeCoups.pop()
    for k in ListeCA[n]:
        ListeCA[k].append(n)

def TestePosition(n):
    global ListeCA, ListeCoups
    Test=OccupePosition(n)
    if Test:
        if len(ListeCoups) == NbCases-1:
            return True
        else:
            LiberePosition()
            return False
    else:
        liste = copy(ListeCA[n])
        for k in liste:
            if TestePosition(k):
                return True
        LiberePosition()
        return False

def valuation(n):
    global ListeCA
    return(len(ListeCA[n]))

def Fusion(A, B):
    C = []
    la = len(A)
    lb = len(B)
    i = 0
    j = 0
    while i < la and j < lb: # on continue tant que les deux listes ne sont pas vides
        if valuation(A[i]) <= valuation(B[j]) :
            C.append(A[i])
            i += 1
        else :
```

```
            C.append(B[j]) ;
            j += 1
    # à ce stade, une (au moins) des deux listes est vide, et on ajoute les éléments de
    if i == la :
        C.extend(B[j:])
    else :
        C.extend(A[i:])
    return C

def TriFusion(L):
    if len(L) == 0:
        return([])
    elif len(L) == 1:
        return([L[0]])
    else:
        return(Fusion(TriFusion(L[0:len(L)//2]),TriFusion(L[len(L)//2:])))

def TriPython(L): # plus rapide!
    return sorted(L, key=valuation)

def TestePosition2(n):
    global ListeCA, ListeCoups
    Test=OccupePosition(n)
    if Test:
        if len(ListeCoups) == NbCases-1:
            return True
        else:
            LiberePosition()
            return False
    else:
        liste=TriFusion(ListeCA[n])
        for k in liste:
            if TestePosition2(k):
                return True
        LiberePosition()
        return False

Init()
TestePosition2(0)
ListeCoups.append(ListeCA[ListeCoups[-1]][0])
# on ajoute la dernière case
print(ListeCoups)
```

```
[0, 17, 32, 49, 59, 53, 63, 46, 61, 55, 38, 23, 6, 12, 2, 8, 25, 40, 57, 51, 34, 24,
```

Rem : on peut s'interroger sur l'intérêt du TriFusion pour des listes de 8 éléments maximum ! Un simple tri par insertion suffit ici (et est même plus rapide).