Algorithmique et Programmation 1 – TD - TP 5
FONCTIONS

Exercice 1 - Carré magique
L'objectif de cet exercice est de vérifier les contraintes des carrés magiques représentés par des listes de listes d'entiers. Par la même occasion, on illustre ici la décomposition d'un problème (relativement) complexe en un certain nombre de sous-problèmes plus simples.

Un carré magique de dimension n est une matrice de taille $n \times n$ contenant des entiers naturels, telle que :
- tous les entiers de 1 à $n \times n$ sont présents dans la matrice (*donc tous les éléments sont distincts*),
- les sommes des entiers de chaque ligne sont toutes égales à une même valeur S,
- la somme de chaque colonne est également S,
- les sommes des deux diagonales de la matrice sont également S.

Par exemple, la matrice 3×3 suivante est un carré magique de dimension 3 :

2	7	6
9	5	1
4	3	8

En Python, nous allons représenter un carré magique par une liste de listes d'entiers : `list[list[int]]`. Pour un carré magique de taille n, chaque ligne de la matrice est représentée par une liste d'entiers. Les lignes sont elle-mêmes stockées dans une liste dans l'ordre séquentiel (la première liste correspond à la première ligne, la seconde liste correspond à la deuxième ligne, etc.).

Voici une expression permettant de construire le carré magique ci-dessus :

```
>>> CarreMagique = [[2, 7, 6],
...                 [9, 5, 1],
...                 [4, 3, 8]]
```

- Les lignes sont : `[2, 7, 6]`, `[9, 5, 1]` et `[4, 3, 8]`
- Les colonnes sont : `[2, 9, 4]`, `[7, 5, 3]` et `[6, 1, 8]`
- Les diagonales sont : `[2, 5, 8]` et `[6, 5, 4]`

1. Définir la fonction `appartient_listeliste` qui étant donné un entier `n` et une liste de listes d'entiers `ll` retourne `True` si l'entier `n` est présent dans la liste `ll`, `False` sinon.
   Par exemple :

```
>>> appartient_listeliste(5, [[1, 2, 3], [4, 5, 6]])
True
>>> appartient_listeliste(7, [[1, 2, 3], [4, 5, 6]])
False
>>> appartient_listeliste(7, CarreMagique)
True
>>> appartient_listeliste(12, CarreMagique)
False
```

2. Définir la fonction `verif_elems` qui, étant donné un entier naturel n non nul et une matrice `mat` d'entiers, retourne `True` si tous les entiers dans l'intervalle $[1, n*n]$ sont présents dans la matrice `mat`, `False` sinon.
   Par exemple :

```
>>> verif_elems(3, CarreMagique)
True
>>> verif_elems(3, [ [2, 7, 6], [8, 5, 1], [4, 3, 8] ])
False
```

3. Définir la fonction `somme_liste` qui retourne la somme des éléments d'une liste d'entiers.
   Par exemple :

```
>>> somme_liste([2, 7, 6])
15
```

4. Définir la fonction `verif_lignes` telle que `verif_lignes(mat, somme)` retourne `True` si toutes les sous-listes de `mat` possèdent la même somme `somme`, `False` sinon.
   Par exemple :

```
>>> verif_lignes(CarreMagique, 15)
True
>>> verif_lignes(CarreMagique, 16)
False
>>> verif_lignes([ [2, 7, 6], [8, 5, 1], [4, 3, 8] ], 15)
False
```

5. Définir la fonction `colonne` qui, étant donné un entier `k` et une matrice `mat` de dimension n, retourne la `k-ième` colonne de la matrice `mat`. On fait l'hypothèse que `k` est compris entre 0 et $n-1$.
   Par exemple :

```
>>> colonne(0, CarreMagique)
[2, 9, 4]
>>> colonne(1, CarreMagique)
[7, 5, 3]
>>> colonne(2, CarreMagique)
[6, 1, 8]
```

6. Définir la fonction `verif_colonnes` telle que `verif_colonnes(mat, somme)` retourne `True` si toutes les colonnes de `mat` possèdent la même somme `somme`, `False` sinon.
   Par exemple :

```
>>> verif_colonnes(CarreMagique, 15)
True
>>> verif_colonnes(CarreMagique, 16)
False
>>> verif_colonnes([ [2, 7, 6], [8, 5, 1], [4, 3, 8] ], 15)
False
```

7. Définir la fonction `diagonale1` (resp. `diagonale2`) qui, étant donné une matrice `mat`, retourne la liste formée des éléments de la première diagonale (resp. seconde diagonale).
   Par exemple :

```
>>> diagonale1(CarreMagique)
[2, 5, 8]
>>> diagonale2(CarreMagique)
[6, 5, 4]
>>> diagonale1([ [ 4, 14, 15, 1], [ 9, 7, 6, 12], [ 5, 11, 10, 8], [16, 2, 3, 13]])
[4, 7, 10, 13]
>>> diagonale2([ [ 4, 14, 15, 1], [ 9, 7, 6, 12], [ 5, 11, 10, 8], [16, 2, 3, 13]])
[1, 6, 11, 16]
```

8. Définir (enfin !) la fonction `verif_magique` qui, étant donné une matrice `mat` de dimension n, retourne `True` si et seulement si elle représente un carré magique.
   Par exemple :

```
>>> verif_magique(CarreMagique)
True
>>> verif_magique([[2, 7, 6],
...                [8, 5, 1],
...                [4, 3, 8]])
False
>>> verif_magique([[2, 7, 6],
...                [8, 5, 1],
...                [4, 3, 9]])
False
```

Exercice 2 - Triangle de Pascal

Soit le programme suivant vu en cours, permettant d'afficher le binôme de Newton pour un entier entré par l'utilisateur :

```
1  def factorielle(n):
2      """Int --> Int
3      Fonction retournant la factorielle de n"""
4      assert type(n) is int, "Factorielle d'un entier"
5      fact = 1
6      for i in range(1, n+1) :
7          fact = fact * i
8      return(fact)

10 def coefficient_binomial(n, k):
11     """Int x Int --> Int, avec k <= n
12     Fonction retournant le coefficient binomial de k et n"""
13     assert type(n) is int, "n est un entier"
14     assert type(k) is int, "k est un entier"
15     #Pour k in [0, n], le coefficient binomial est un entier
16     coeff = factorielle(n)//(factorielle(k) * factorielle(n-k))
17     return(coeff)

19 def newton(n):
20     """Int --> None, n != 0
21     (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k)."""
22     assert type(n) is int, "n est un entier"
23     assert not(n == 0), "n différent de 0"
24     #Initialisation de la chaine
25     chaine = "(a + b)^" + str(n) + " = "
26     for k in range(n+1) :
27         c = coefficient_binomial(n, k)
28         p = n - k
```

```
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
        if not(p == 0) : #si un facteur = 0, on écrit pas les autres facteurs
            chaine = chaine + "a"
            if not(p == 1) :
                chaine = chaine + "^" + str(p)
        if not(k == 0):
            chaine = chaine + "b"
            if not(k == 1) :
                chaine = chaine + "^" + str(k)
        if k < n: #On continue avec + sauf pour la dernière occurence
            chaine = chaine + " + "
    print(chaine)

n = int(input("Calculer le binôme de newton pour n = "))
newton(n)
```

1. Récupérer le fichier `newton.py` sur Moodle, et le tester.

Le **triangle de Pascal** est une présentation des coefficients binomiaux dans un triangle. La construction du triangle est régie par la relation de Pascal, pour tout entiers n et k tels que $0 < k < n$:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Par exemple, pour $n = 8$:

```
Afficher le triangle de Pascal jusquà n = 8
[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]
[1, 8, 28, 56, 70, 56, 28, 8, 1]
```

2. Construire, dans une liste de listes, le triangle de Pascal jusqu'à un entier n entré par l'utilisateur en utilisant les fonctions écrites pour calculer le binôme de Newton : dans le triangle de Pascal, une même ligne contient tous les coefficients intervenant dans le développement d'une puissance de la somme de deux termes. Affichez ensuite ce triangle
3. Construire le triangle de Pascal jusqu'à un entier n entré par l'utilisateur en utilisant la formule de Pascal [1] :

   On en déduit une méthode de construction du triangle de Pascal, qui consiste, sous forme pyramidale, à placer 1 au sommet de la pyramide, puis 1 et 1 en dessous, de part et d'autre. Les extrémités des lignes sont toujours des 1, et les autres nombres sont la somme des deux nombres directement au-dessus. Sous forme triangulaire, i étant l'indice de ligne et j l'indice de colonne :
   - *placer dans la colonne 0 des 1 à chaque ligne, et des 1 à chaque entrée de la diagonale,*
   - *en partant du haut et en descendant, compléter le triangle en ajoutant deux coefficients adjacents d'une ligne, pour produire le coefficient de la ligne inférieure, en dessous du coefficient de droite.*

1. Source : `https://fr.wikipedia.org/wiki/Triangle_de_Pascal`