DS 3 d'informatique - MPSI - Carrés magiques - Correction

Question 1.a – Pour $n = 1$, il y a un seul carré magique normal.

En effet, la seule possibilité est de mettre l'entier 1 dans la seule case de la grille. On obtient bien un carré magique normal.

Question 1.b – Pour $n = 2$, il n'y a pas de carré magique normal.

En effet, les sommes des lignes et des colonnes devraient être égales à $2\dfrac{4+1}{2} = 5$. Ainsi, si on s'intéresse à la case c contenant l'entier 1, alors la case se trouvant sur la même ligne que c doit contenir 4 et la case se trouvant sur la même colonne que c doit aussi contenir 4. C'est impossible car 4 ne doit apparaître qu'une fois dans le carré magique.

Question 2.a – Pour $a = 1$, $b = 3$ et $c = 5$, on obtient un carré magique normal :

6	1	8
7	5	3
2	9	4

En particulier, le triplet (a, b, c) vérifie bien la condition (C).

Pour trouver les valeurs de a, b, c, on peut raisonner ainsi (ce raisonnement n'était pas attendu sur la copie) :

- → Comme la somme des lignes/colonnes/digonales dans une grille de Lucas est égale à $3c$ et que dans un carré magique de taille 3, ces sommes sont égales à $3\dfrac{9+1}{2} = 15$; on a nécessairement $c = 5$.
- → À cause de la condition (C), on a $1 \leqslant a < b \leqslant 4$ et $b \neq 2a$. Les seules valeurs possibles pour (a, b) sont $(1, 3)$, $(1, 4)$, $(2, 3)$, $(3, 4)$.
- → Solution 1 : on teste les 4 possibilités.
- → Solution 2 : le plus grand entier dans la grille de Lucas est $a + b + c$, le plus grand entier dans un carré magique de taille 3 est 9. Donc $a + b = 9 - c = 4$. Finalement, la seule possibilité est $(1, 3, 5)$.

Question 2.b –

```
def Lucas(a, b, c):
    assert 0 < a and a < b and b < c - a and b != 2*a
    return [
        [c+a, c-a-b, c+b],
        [c-a+b, c, c+a-b],
        [c-b, c+a+b, c-a]
    ]
```

Question 3.a –

```
def testTaille(G):
    n = len(G)
    for L in G:
        if len(L) != n:
            return False
    return True
```

Question 3.b – La signature de `testTaille` est :

`testTaille(G: grille) -> bool`

Question 4 –

```
def testVal1(G):
    n = len(G)
    for L in G:
        for e in L:
            if e < 1 or e > n*n:
                return False
    return True
```

Question 5 – On remarque que dans cette procédure, il y a au plus n^2 tours de boucle. De plus, chaque tour s'exécute en temps $\mathcal{O}(n^2)$ car il faut parcourir tous les éléments de `G` Finalement :

Le temps d'exécution de la procédure est en $\mathcal{O}(n^4)$

Question 6.a –

```
1  def getT(G):
2      n = len(G)
3      T = [False] * (n*n)
4      for L in G:
5          for i in L:
6              if 1 <= i and i <= n*n:
7                  T[i-1] = True
8      return T
```

Question 6.b –

```
1  def testVal2(G):
2      for b in getT(G):
3          if not b:
4              return False
5      return True
```

Question 7 – Le temps d'exécution de `testVal` est en $\mathcal{O}(n^2)$

En effet, la fonction `getT` s'exécute en temps $\mathcal{O}(n^2)$:

→ La ligne 3 s'exécute en temps $\mathcal{O}(n^2)$ et toutes les autres lignes s'exécutent en temps constant.

→ Les lignes 6 et 7 sont exécutées une fois pour chaque élément de `G`, c'est à dire n^2 fois.

La fonction `testVal2` s'exécute en temps $\mathcal{O}(n^2)$:

→ L'appel à `getT` s'exécute en temps $\mathcal{O}(n^2)$ et toutes les autres lignes s'exécutent en temps constant.

→ Les lignes 3 et 4 sont exécutées une fois pour chaque élément de `T`, c'est à dire n^2 fois.

Question 8.a –

```
1  def testL(G):
2      n = len(G)
3      N = (n*(n*n+1))//2
4      for i in range(n):
5          s = 0
6          for j in range(n):
7              s += G[i][j]
8          if s != N:
9              return False
10     return True
```

Question 8.b – Pour tester la condition (C) :

Il suffit de remplacer `G[i][j]` par `G[j][i]` à la ligne 7.

Question 9.a –

Coin	Haut gauche	Haut droit	Bas droit	Bas gauche
Coordonnées dans `G1`	$(0,0)$	$(0,n-1)$	$(n-1,n-1)$	$(n-1,0)$
Coordonnées dans `G2`	$(0,n-1)$	$(n-1,n-1)$	$(n-1,0)$	$(0,0)$

Question 9.b –

```
def rot(G1):
    n = len(G1)
    G2 = [[None]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            G2[j][n-1-i] = G1[i][j]
    return G2
```

Question 9.c –

```
def testC(G):
    return testL(rot(G))
```

Question 10.a – La case de coordonnées (i,j) est sur la diagonale si et seulement si $\boxed{i=j}$.

Question 10.b –

```
def testD1(G):
    n = len(G)
    N = (n*(n*n+1))//2
    s = 0
    for i in range(n):
        s += G[i][i]
    return s == N
```

Question 11.a – La case de coordonnées (i,j) est sur l'anti-diagonale si et seulement si $\boxed{j=n-1-i}$.

Question 11.b –

```
def testD2_aux(G, i, s):
    n = len(G)
    if i == n:
        return s == (n*(n*n+1))//2
    else:
        return testD2_aux(G, i+1, G[i][n-1-i] + s)

def testD2(G):
    return testD2_aux(G, 0, 0)
```

Question 12 –

```
def estCMN(G):
    return testL(G) and testC(G) and testD1(G) and testD2(G)
```

Question 13 –

```
def siamoise(n):
    assert n % 2 == 1
    G = [[None for _ in range(n)] for _ in range(n)]
    i = 0; j = n//2
    for k in range(1, n*n+1):
        G[i][j] = k
        i2 = (i-1) % n; j2 = (j+1) % n
        if G[i2][j2] is None:
            i = i2; j = j2
        else:
            i = (i+1) % n
    return G
```

Question 14.a –

```
def f1(G, m):
    n = len(G)
    return [[G[i % n][j % n] for j in range(n*m)] for i in range(n*m)]
```

Question 14.b –

```
def f2(G, m):
    n = len(G)
    return [[G[i // m][j // m] for j in range(n*m)] for i in range(n*m)]
```

Question 15 –

```
def sous(G, m):
    n = len(G)
    return [[G[i][j] - m for j in range(n)] for i in range(n)]

def mult(G, m):
    n = len(G)
    return [[m*G[i][j] for j in range(n)] for i in range(n)]

# Suppose G1 et G2 ont la même taille
def somme(G1, G2):
    assert len(G1) == len(G2)
    n = len(G1)
    return [[G1[i][j] + G2[i][j] for j in range(n)] for i in range(n)]
```

```
def produit(G1, G2):
    n1 = len(G1)
    n2 = len(G2)
    H1 = f1(G1, n2)
    H2 = f2(mult(sous(G2 , 1), n1*n1), n1)
    return somme(H1, H2)
```

Question 16 – Chaque grille `G` sera accompagnée d'une liste « `M: grille` » contenant les entiers de $[\![1, n^2]\!]$ qui n'apparaissent pas dans `G`. Par exemple, si `G` est la grille de la figure 5, alors `M = [7, 18, 23]`.

Soit `G0` une grille partiellement remplie et m le nombre de cases vides dans `G0`. La procédure permettant de générer toutes les complétions possibles de `G0` va construire des listes L_0, L_1, ..., L_m ayant pour type :

`list[grille, list[int]]`

Pour chaque $k \in [\![0, m]\!]$, la liste L_k va contenir tous les couples `(G, M)` où `G` est obtenu à partir de `G0` en complétant les e premières cases vides et `M` est la liste décrite dans le paragraphe précédent. Par exemple si `G0` est la liste de la figure 5, alors :

- ⋆ L_0 ne contient que le couple `(G0, M(G0))`.
- ⋆ L_1 contient trois grilles : la grille `G0` dans laquelle la case $(0, 0)$ a été remplie par 7, 18 ou 23.
- ⋆ L_2 contient six grilles : les grilles des figures 6 à 11 dans lesquelles la case d'indice $(0, 2)$ a été effacée.
- ⋆ L_3 contient six grilles : les grilles des figures 6 à 11.

La fonction intermédiaire `etape` prend en entrée une liste L_k ainsi que les coordonnées (i, j) de la prochaine case à compléter et renvoie la liste L_{k+1}

```
def etape(L1, i, j):
    L2 = []
    for (G1, M1) in L1:
        for k in range(len(M1)):
            G2 = [G1[i][:] for i in range(len(G1))]
            G2[i][j] = M1[k]
            M2 = M1[:k] + M1[k+1:]
            L2.append((G2, M2))
    return L2
```

```
def genCMN(G0):
    n = len(G0)
    T = getT(G0)
    M = [i for i in range(1, n*n+1) if not T[i-1]]
    L = [(G0,M)]
    for i in range(n):
        for j in range(n):
            if G0[i][j] == 0:
                L = etape(L, i, j)
    return [G for (G,_) in L if estCMN(G)]
```