Informatique MPSI - Devoir surveillé n°4 - Samedi 25/05/2024 - 2 heures 30
Calculatrices interdites.

Les programmes demandés seront écrits en Python. **Le travail devra bien sûr être soigné, les codes clairs et commentés autant que vous le jugez nécessaire. Veillez à réserver du temps, après la rédaction de chaque programme, pour le vérifier pour éviter les erreurs de syntaxe.**

Dans chaque exercice, il est bien évidemment possible de faire appel à des fonctions ou procédures créées dans les questions antérieures.

Le sujet comporte **8 pages**. Il est composé d'un exercice et d'un problème indépendants qui seront traités sur des **copies séparées**. La fin du problème n'est pas à traiter, elle n'est donnée que pour terminer la résolution du problème a posteriori pour ceux qui le souhaitent (ou qui seraient très rapides).

Rappels concernant le langage Python : il n'est pas possible d'utiliser des fonctions internes à Python sur les listes ou les chaines de caractères telles que `min, max, count, remove, index ....` Seules les instructions basiques telles que `len(liste), liste.append(e)` sont autorisées. Le tranchage (ou slicing), ainsi que la concaténation sont également permis. Il n'est pas utile de rappeler sur la copie la signature des fonctions demandées dans les différentes questions.

Exercice. Représentation de Zeckendorf

Dans cet exercice, on s'intéresse à la célèbre suite de Fibonacci qu'on utilisera pour encoder des nombres, puis pour implémenter une exponentiation rapide.

Suite de Fibonacci.

La suite de Fibonacci $(\mathcal{F}_k)_{k\in\mathbb{N}}$ est définie par :

$$\begin{cases} \mathcal{F}_0 = 1, & \mathcal{F}_1 = 2 \\ \forall k \in \mathbb{N}, & \mathcal{F}_{k+2} = \mathcal{F}_{k+1} + \mathcal{F}_k \end{cases}$$

1. Donner les 10 premiers termes de la suite de Fibonacci.

2. Écrire une fonction `fibo` qui prend en entrée un entier k $\in \mathbb{Z}$ et renvoie la liste $[\mathcal{F}_0, \mathcal{F}_1, \dots, \mathcal{F}_{\mathrm{k}-1}]$.
Si k ≤ 0, votre fonction déclenchera une erreur.

Représentation de Zeckendorf.

Dans la suite, on dira qu'une somme d'entiers est valide si les quatre conditions suivantes sont respectées :

- Chaque terme de la somme est un élément de la suite de Fibonacci.
- Ces termes sont strictement décroissants.
- Si on note $a \in \mathbb{N}^*$ la valeur de la somme et k l'entier tel que $\mathcal{F}_k \leqslant a < \mathcal{F}_{k+1}$, alors $\mathcal{F}_k$ apparait dans la somme.
- Pour tout $k \in \mathbb{N}$, si $\mathcal{F}_k$ apparait dans la somme alors $\mathcal{F}_{k+1}$ n'y apparait pas.

On admettra que tout entier strictement positif $a \in \mathbb{N}^*$ peut s'écrire comme une unique somme valide.
Par exemple, avec $a = 40$, on obtient $a = 40 = 34 + 5 + 1 = \mathcal{F}_7 + \mathcal{F}_3 + \mathcal{F}_0$.

3. Écrire $a = 60$ comme une somme valide.

À chaque entier $a \in \mathbb{N}^*$ on associe une chaine de caractères `s` contenant une suite de 0 et de 1 appelée *représentation de Zeckendorf* de a. Pour obtenir `s`,

- On écrit a comme une somme valide, puis on définit `s` comme étant la chaine de caractères telle que :
  - Si $\mathcal{F}_{\mathtt{k}}$ apparait dans la somme alors `s[k]` vaut "1".
  - Sinon, `s[k]` vaut "0".
- Afin de garantir l'unicité, on impose que le dernier caractère de `s` soit un 1.

Par exemple, la représentation de Zeckendorf de $a = 40$ est `'10010001'`.

4. Quelle est la représentation de Zeckendorf de $a = 60$?

5. Écrire une fonction `zeckToInt` qui prend en entrée une chaine de caractères `s` et renvoie l'entier a dont `s` est la représentation de Zeckendorf.

6. **a.** Écrire une fonction `tailleZeck` qui prend en entrée un entier a et renvoie la taille (i.e. la longueur) de la représentation de Zeckendorf de a, notée ℓ. On garantira une complexité en $O(\ell)$.

b. Écrire une fonction `intToZeck` qui prend en entrée un entier et renvoie sa représentation de Zeckendorf.

Code de Fibonacci.

On souhaite maintenant encoder une suite de nombres $a_0, a_1, \ldots, a_{n-1} \in \mathbb{N}^*$ par une suite de 0 et de 1 appelée *code de Fibonacci*.
Pour cela, on écrit la représentation de Zeckendorf de a_0, puis on y concatène un 1, puis on y concatène la représentation de Zeckendorf de a_1, puis on y concatène un $1, \ldots$, puis on y concatène la representation de Zeckendorf de a_{n-1} et enfin on y concatène un 1.
Par exemple, le code de Fibonacci de la suite 5, 40, 21 est 00011100100011000000011.
En Python, la suite de nombres $a_0, a_1, \ldots, a_{n-1}$ sera représentée par une liste `A: list[int]` de taille `n` et le code de Fibonacci correspondant sera représenté par une chaine de caractères `CF`.

7. **a.** Écrire une fonction `AtoCF` qui prend en entrée `A` et renvoie `CF`.

b. Écrire une fonction `CFtoA` qui prend en entrée `CF` et renvoie `A`. On expliquera succinctement la procédure utilisée.

Exponentiation rapide.

Le but de cette question est d'utiliser la représentation de Zeckendorf d'un entier $a \in \mathbb{N}^*$ pour calculer rapidement x^a pour $x \in \mathbb{R}$.

8. Sans utiliser l'opérateur **, écrire une fonction `puiss` qui prend en entrée une chaine de caractères `s` contenant la représentation de Zeckendorf d'un entier $a \in \mathbb{N}^*$, ainsi qu'un flottant $x \in \mathbb{R}$, et renvoie x^a. Votre fonction devra manipuler directement `s`, sans jamais calculer explicitement la valeur de a, en effectuant le minimum de multiplications possibles. On expliquera succinctement la procédure utilisée.

Problème. Jeu de taquin

Dans tout le problème `n` désigne un entier naturel strictement positif, supposé défini comme une variable globale. Toutes les complexités devront être exprimées en fonction de ce `n`.
On pourra utiliser sans restriction la fonction `copy` ci-dessous :

```
def copy(G):
    ''' 'G: list[list[int]]
    Returns :  list[list[int]]' ' '
    return [L[:]  for L in G]
```

1 Introduction

Le taquin est un jeu de réflexion composé d'un cadre contenant `n` lignes et `n` colonnes. Parmi les $\mathtt{n}^2$ cases de ce cadre, une case est vide et les autres contiennent des plaques numérotées de 1 à $\mathtt{n}^2 - 1$. Les figures 1 à 5 présentent des grilles possibles pour le jeu du taquin dans le cas où $\mathtt{n} = 4$.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 1

5	11	1	2
14	10	4	7
3	13		12
9	8	6	15

Figure 2

5	11	1	2
14	10	4	7
3		13	12
9	8	6	15

Figure 3

5	11	1	2
14	10	4	7
3	8	13	12
9		6	15

Figure 4

5	11	1	2
14	10	4	7
3	8	13	12
9	6		15

Figure 5

Le but du joueur est de déplacer les plaques de telle manière que :

- La case vide soit en bas à droite.
- Les numéros des plaques soient dans l'ordre croissant lorsqu'on les lit ligne par ligne.

Par exemple, dans le cas $\mathtt{n} = 4$, l'objectif est d'obtenir la grille de la figure 1. À chaque tour de jeu, les plaques peuvent bouger en glissant les unes par rapport aux autres dans le cadre. Ainsi, pour la grille de la figure 2, il y a quatre mouvements possibles : faire glisser la plaque 4, 6, 13 ou 12 vers la case vide. Si le joueur choisit de faire glisser la case 13, il obtient la grille de la figure 3 où il y a de nouveau quatre mouvements possibles. Pour passer de la figure 3 à la figure 4, il faut faire glisser la case 8 vers le haut. De même, pour passer de la figure 4 à la figure 5, il faut faire glisser la case 6 vers la gauche. Ainsi, à chaque étape, il y a deux, trois ou quatre mouvements possibles en fonction de la position de la case vide (deux mouvements pour la figure 1, trois pour les figures 4 et 5, et quatre pour les figures 2 et 3).

2 Représentation en Python

Étant donnée une grille de taquin avec `n` lignes et `n` colonnes, chaque case est repérée par un couple d'entiers $(\mathtt{i}, \mathtt{j}) \in [\![0\,;\,\mathtt{n}-1]\!]^2$ appelés les coordonnées de la case. L'indice $\mathtt{i} = 0$ correspond à la ligne du haut, $\mathtt{i} = \mathtt{n} - 1$ à la ligne du bas, $\mathtt{j} = 0$ à la colonne de gauche et $\mathtt{j} = \mathtt{n} - 1$ à la colonne de droite. En machine, une grille est représentée par une liste de listes d'entiers `G` telle que pour tout $(\mathtt{i}, \mathtt{j}) \in [\![0\,;\,\mathtt{n}-1]\!]^2$:

$$\mathtt{G[i][j]} = \begin{cases} \mathtt{0} \text{ si la case de coordonnées } (\mathtt{i}, \mathtt{j}) \text{ est vide} \\ \mathtt{x} \text{ si la case de coordonnées } (\mathtt{i}, \mathtt{j}) \text{ contient la plaque numéro } \mathtt{x} \end{cases}$$

Par exemple, les grilles des figures 1, 2 et 5 correspondent respectivement à `GFig1, GFig2` et `GFig5` avec :

```
Gfig1   =   [[1, 2, 3, 4],  [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 0]]
Gfig2   =   [[5, 11, 1, 2],  [14, 10, 4, 7], [3, 13, 0, 12], [9, 8, 6, 15]]
Gfig5   =   [[5, 11, 1, 2],  [14, 10, 4, 7], [3, 8, 13, 12], [9, 6, 0, 15]]
```

1. Dans cette question, on note `G` une liste de listes d'entiers.

Écrire une fonction de signature `test(G: list[list[int]]) -> bool` qui renvoie `True` si `G` est une liste de `n` sous-listes de taille `n` et si tous les entiers présents dans `G` appartiennent à $\left[\!\left|0\,;\,\mathtt{n}^2-1\right|\!\right]$ et `False` dans le cas contraire.

Jusqu'à la fin du sujet, on pourra supposer sans le vérifier que `test(G)` s'évalue à `True`. On dira que `G` est *bien formée* si tous les entiers de $\left[\left|0\,;\,\mathtt{n}^2-\mathtt{1}\right|\right]$ sont présents dans G.

2. À l'aide de boucles `while` et sans utiliser de boucle `for`, écrire une fonction de signature

`coord(G : list[list[int]], x : int) -> (int, int)` ou `NoneType`

qui renvoie le couple `(i, j)` tel que `G[i][j]` vaut `x`, ou renvoie `None` si `x` est absent de `G`.

3. Afin de tester si `G` est bien formée, on définit la fonction `tousPresents` ci-dessous. Donner la signature et la complexité de la fonction `tousPresents`. Pour la complexité, on attend une justification.

```
def tousPresents(G):
    n = len(G)
    for x in range(n*n):
        if coord(G,x) == None:
            return False
    return True
```

4. La complexité de la fonction `tousPresents` n'étant pas optimale, on cherche à l'améliorer. Expliquer comment tester si `G` est bien formée avec une complexité quadratique en `n`. Votre réponse doit être une description en français, pas un code Python.

Jusqu'à la fin du sujet, on pourra supposer sans le vérifier que `G` est bien formée.

3 Mouvements au taquin

À chaque tour de jeu, le joueur fait glisser l'une des plaques vers la case vide. Il s'agit donc d'un mouvement vers le haut, vers le bas, vers la gauche ou bien vers la droite. On représentera ces quatre possibilités par une chaine de caracteres `M` $\in$ {`'H'`, `'B'`, `'G'`, `'D'`} (pour Haut, Bas, Gauche, Droite). *Par exemple*, pour passer de la figure 2 à la figure 5, en passant par les figures 3 et 4, on effectue successivement les mouvements `'D'`, `'H'`, puis `'G'`.

5. **a.** On note `(i0, j0)` les coordonnées de la case vide de la grille.

i. A quelle condition sur `i0, j0` et `n` le mouvement `'H'` est-il possible ? Même question pour les mouvements respectifs `'B'`, `'D'` et `'G'`.

ii. Écrire une fonction de complexité constante

`estLicite(i0: int, j0: int, n: int, M: str) -> bool`

qui renvoie `True` si le mouvement `M` est possible et `False` sinon. Votre fonction renverra également `False` dans le cas où `M` $\notin$ {`'H'`, `'B'`, `'G'`, `'D'`}.

Par exemple :

M	'H'	'B'	'G'	'D'
i0=3,j0=3,n=4 (Figure 1)	False	True	False	True
i0=2,j0=2,n=4 (Figure 2)	True	True	True	True
i0=3,j0=2,n=4 (Figure 5)	False	True	True	True

b. Ecrire une fonction de signature

`mvt(G1: list[list[int]],M: str) -> list[list[int]]`

qui prend en entrée une grille de taquin `G1` et renvoie la grille `G2` obtenue après avoir effectué le mouvement `M`.

Votre fonction déclenchera une erreur si la fonction de la question précédente indique que le mouvement n'est pas possible.

Votre fonction devra commencer par la ligne `G2 = copy(G1)` et ne pas modifier la grille `G1` donnée en entrée.

Par exemple: `mvt(GFig2, 'D')` est la grille de la figure 3. et `mvt(GFig5, 'D')` est la grille de la figure 4.

c. Donner en la justifiant la complexité de la fonction `mvt` en fonction de `n`.

Au taquin, le but est d'obtenir une grille particulière appelée grille finale et notée `GF` dans la suite. Pour générer cette grille, on écrit deux fonctions :

```
1  def makeGF1(n):
2      GF = []
3      k = 1
4      for _ in range(n):
5          GF.append([])
6          for _ in range(n):
7              GF[-1].append(k)
8              k += 1
9      GF[-1][-1] = 0
10     return GF
```

```
1  def makeGF2(n):
2      GF = f(n)
3      for i in range(n):
4          for j in range(n):
5              GF[i][j] = g(i,j,n)
6      return GF
```

6. **a.** Lors d'un appel à la fonction `makeGF1` avec l'argument `n` $= 3$:

i. Quelle est la valeur de la variable `GF` à la fin de chacun des trois tours de la boucle `for` située ligne 4?

ii. Quelle est la liste renvoyée?

b. On souhaite faire en sorte que la fonction `makeGF2` renvoie la même liste que la fonction `makeGF1`. Définir les fonctions `f` et `g` appelées lignes 2 et 5 de `makeGF2`. Le code de la fonction `f` devra faire une ligne (sans compter la ligne avec le `def`) et contenir uniquement des compréhensions de liste. La fonction `g` devra s'exécuter en temps constant.

7. Soit `LM: list[str]` une liste de mouvements (c'est à dire une liste dont chaque élément est `'H'`, `'B'`, `'G'` ou `'D'`). Ecrire une fonction de signature

```
verif(G1:  list[list[int]], G2:  list[list[int]], LM: list[str]) -> bool
```

qui indique si la liste de mouvements `LM` permet de passer de la grille `G1` à la grille `G2`. La fonction ne devra pas modifier les grilles `G1` et `G2` données en argument.

Par exemple : `verif(GFig2, GFig5, ['D', 'H', 'G'])` vaut `True`

4 Configurations solubles

Certaines grilles de taquin ne peuvent pas être résolues (dans le sens où aucune suite de mouvements ne permet d'obtenir la grille finale définie dans la partie 3). Par exemple, si dans la figure 1, on échange les plaques numérotées 1 et 2 alors il n'est pas possible de retrouver la grille de la figure 1. Pour tester si une grille peut être résolue, on appelle `L(G)` la liste contenant les entiers de la grille lorsque celle-ci est lue ligne par ligne. Par exemple pour la figure 2, on obtient :

```
L(G) = [5, 11, 1, 2, 14, 10, 4, 7, 3, 13, 0, 12, 9, 8, 6, 15]
```

Étant donnée une liste `G`, on appelle nombre caractéristique de `G` l'entier `c = d + i0 + j0` où :

- `d` est le nombre de couples (k_1, k_2) tels que $k_1 < k_2$ et $L(G)[k_1] > L(G)[k_2]$.
- `i0` et `j0` sont les deux entiers tels que `G[i0][j0]=0`.

Théorème (admis). Soient G et G′ deux grilles et c, c' leurs nombres caractéristiques alors :

$$\text{ll est possible de passer de G à G}' \text{ par une suite de mouvements} \iff c \equiv c' \quad [2]$$

8. **a.** Écrire une fonction `nbCarac` qui prend en entrée une liste de listes d'entiers `G` et renvoie son nombre caractéristique.

b. Quelle est la complexité de `nbCarac` en fonction de `n`? Justifier.

9. A l'aide du théorème :

a. Montrer que la grille de la figure 6 ne peut pas être résolue.

2	1	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 6

b. Écrire une fonction `testParite` qui prend en entrée deux grilles `G1`, `G2` et renvoie un booléen indiquant s'il est possible de passer de l'une à l'autre par une suite de mouvements.
La comparaison de deux listes de listes à l'aide == est autorisée.

5 Résolution

Dans cette partie, on présente une méthode de résolution pour le jeu du taquin.
On a pour cela besoin d'une procédure intermédiaire permettant de placer la case vide à côté d'une plaque dont le numéro `x` est donné. Afin de ne pas bouger des plaques placées précédemment, la procédure se déroulera en deux étapes comme suit :

- La case vide est d'abord déplacée dans le coin inférieur droit de la grille en utilisant uniquement les mouvements `'H'` et `'G'`.
- La case vide est ensuite déplacée sur une case adjacente à la plaque `x` en utilisant uniquement des mouvements `'B'` et `'D'`. Pour cela, on distinguera deux cas :
  - Si la plaque `x` se trouve sur la dernière ligne, alors on place la case vide sur la case se trouvant à droite de `x`.
  - Sinon, on place la case vide sur la case se trouvant en dessous de `x`.

10. Écrire une fonction de signature

```
deplVide(G: list[list[int]], x:  int) -> (list[list[int]], list[str])
```

qui déplace la case vide à côté de la plaque numérotée par `x` en suivant la procédure décrite ci-dessus. Votre fonction renverra la grille obtenue ainsi que la liste des mouvements à effectuer. La complexité devra être de l'ordre de n^3.

Soit G_0 une grille et $k \in \mathbb{N}$ un entier. On cherche à générer toutes les grilles atteignables à partir G_0 en utilisant k mouvements. On note L_k la liste contenant tous les couples de la forme (`G, LM`) avec:

- `LM` une liste de longueur k dont chaque élément est `'H'`, `'B'`, `'G'` ou `'D'`. Dans un soucis d'optimisation, on interdit à deux mouvements successifs de s'annuler. Formellement, pour tout $i \in [|0\,;\,k-2|]$:

$$(LM[i], LM[i+1]) \notin \{('H','B');('B','H');('G','D');('D','G')\}.$$

- `G` la grille obtenue à partir de G_0 en appliquant la suite des mouvements contenus dans `LM`.

Afin de prendre en compte ces contraintes on définit le dictionnaire suivant (variable globale) :
`mvt_interdit = {'H': 'B'; 'B': 'H'; 'G': 'D'; 'D':'G'}`

11. Supposons que la liste L_k ait déjà été générée. Écrire une fonction `une_etape` qui prend en entrée L_k et renvoie L_{k+1}.

La suite du sujet n'est à traiter que si tout le reste a été abrodé. Dans le cas contraire elle ne sera pas corrigée.

Une stratégie envisageable pour résoudre le problème du taquin serait de partir de la grille initiale et de générer les listes L_0, L_1, L_2, ... jusqu'à ce que l'une d'entre elles contienne la grille finale. Malheureusement, cette méthode est beaucoup trop lente pour pouvoir être utilisée en pratique. On va donc procéder autrement. L'idée est de placer correctement les plaques une à une, en veillant à ne pas bouger (ou à remettre en place) les plaques qui ont déjà été correctement placées. On introduit pour cela une liste `LC` appelée *liste de contraintes*. Chaque élément d'une liste de contraintes est un triplet d'entiers (`i,j,x`). Une grille `G` vérifiera la liste de contraintes `LC` si, pour tout (`i,j,x`) appartenant à `LC`, la plaque numéro `x` se trouve sur la case de coordonnées (`i, j`) de la grille `G`.
Par exemple, si `LC = [(0,0,1),(0,1,2),(0,3,3),(1,3,4)]`, une grille qui vérifie `LC` sera comme dans la figure 7 (une case avec un point pouvant être vide ou bien contenir une plaque quelconque).

12. Ecrire une fonction `verif_contraintes` qui prend en entrée une grille `G` et une liste de contraintes `LC` et renvoie le booléen indiquant si `G` vérifie les contraintes de `LC` ou pas.

13. Ecrire une fonction de signature

```
atteindre(i  :  int,j:int,x:int,G:list[list[int]],LM:list[str],LC:list[tuple])
          -  >  (G1:list[list[int]],LM1:list[str])
```

qui prend en entrée trois entiers `i, j, x`, une grille `G`, une liste `LM` contenant les mouvements déjà effectués, et une liste de contraintes `LC` et qui renvoie un couple `(G1, LM1)` où `G1` est une grille, obtenue à partir de `G`, respectant les contraintes `LC` et dans laquelle `x` se trouve à la position `(i,j)`, et `LM1` le chemin à parcourir (en tenant compte de `LM`).

1	2	.	3
.	.	.	4
.	.	.	.
.	.	.	.

Figure 7

1	2	but3	but2
.	.	.	but1
.	.	.	3
.	.	.	.

Figure 8

1	2	.	but6
.	.	.	but5
.	.	.	but4
4	but1	but2	but3

Figure 9

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	18	20	22	24
17	19	21	23	

Figure 10

Soit (`i,j,x`) trois entiers et `LC` une liste de contraintes. On souhaite déplacer la plaque numéro `x` vers la case de coordonnées (`i,j`) tout respectant les contraintes de `LC`. Voici la procédure à respecter :

- On transfère la case vide à côté de `x` (voir question 10),
- Puis on bouge la plaque `x` de proche en proche jusqu'à ce qu'elle ait atteint sa position finale en respectant le protocole suivant : on note (i_x, j_x) les coordonnées initiales de la plaque `x` (après déplacement de la case vide) et on distingue deux cas:
  - Si $j_x > j$, on commence par déplacer la plaque x en remontant le long de la colonne j_x jusqu'à la case (i, j_x) (on cherche à atteindre successivement ($i_x - 1$, j_x) puis ($i_x - 2$, j_x), ...,(i, j_x)), puis on se déplace vers la gauche sur la ligne `i` jusqu'à la case (i, j) (on cherche à atteindre successivement (i, $j_x - 1$) puis (i, $j_x - 2$), ..., (i, j)). La figure 8 est un exemple où `i=0,j=2,x=3,`i_x`=2` et j_x`=3`. Notez que sur cette figure, chaque but ne représente pas un seul mouvement, mais une suite de mouvements à déterminer grâce à la fonction `atteindre`.
  - Si $j_x \leqslant j$, on commence par déplacer la plaque vers la droite jusqu'à la case (i_x,j), puis vers le haut jusqu'à la case (`i,j`). La figure 9 est un exemple où `i=0,j=3,x=4,`i_x`=3` et j_x`=0`.

14. Ecrire une fonction `deplace` qui a les mêmes arguments que la fonction `atteindre` et qui renvoie un couple `(G1, LM1)` où `G1` est une grille, obtenue à partir de `G` respectant les contraintes `LC` et dans laquelle `x` se trouve à la position `(i,j)`, et `LM1` le chemin à parcourir (en tenant compte de `LM`) en respectant la procédure ci-dessus.

Il ne reste plus qu'à placer les plaques les unes après les autres dans la grille. Pour cela, on commence par remplir les $n - 2$ premières lignes dans l'ordre, puis les deux dernières lignes, colonne par colonne. Par exemple pour $n = 5$, la figure 10 indique l'ordre à suivre. Notez que contrairement aux autres figures, les nombres sur la figure 10 ne représentent pas les numéros des plaques, mais l'ordre dans lequel il faut les placer.

15. Ecrire une fonction de signature `resolution(G: list[list[int]]) -> (list[str]` ou `Nonetype)` qui renvoie la liste des mouvements à effectuer pour passer de `G` à la grille finale (voir question 6). Cette fonction doit renvoyer `None` dans le cas où il n'y a pas de solution (voir partie 4).