Problème 1. Résolution du Sudoku

Notations. Pour m et n deux entiers naturels, $[\![m;n]\!]$ désigne l'ensemble des entiers k tels que $m \leq k \leq n$.

La résolution d'une grille de Sudoku est une gymnastique du cerveau qui peut être assimilée à un décodage correcteur d'effacement. En effet, à partir d'une grille presque vide, il est possible (pour une grille bien faite) de la compléter d'une unique manière.
L'objectif de cet exercice est de mettre en œuvre deux méthodes permettant de compléter une grille de Sudoku, l'une naïve, et l'autre par backtracking.

Une grille de Sudoku est une grille de taille 9×9, découpée en 9 carrés de taille 3×3. Le but est de la remplir avec des chiffres de $[\![1;9]\!]$, de sorte que chaque ligne, chaque colonne et chacun des 9 carrés de taille 3×3 contienne une et une seule fois chaque entier de $[\![1;9]\!]$. On dira alors que la grille est complète. En pratique, certaines cases sont déjà remplies et on fera l'hypothèse que le Sudoku qui nous intéresse est bien écrit, c'est-à-dire qu'il possède une unique solution.
On représente en Python une grille de Sudoku par une liste de taille 9×9, c'est-à-dire une liste de 9 listes de taille 9, dans laquelle les cases non remplies sont associées au chiffre 0. Ainsi, la grille suivante est représentée par la liste ci-contre :

	6					2		5
4			9	2	1			
	7				8			1
					5			9
6	4						7	3
1			4					
3			7				6	
			1	4	6			2
2		6					1	

```
L = [[0,6,0,0,0,0,2,0,5],[4,0,0,9,2,1,0,0,0],
     [0,7,0,0,0,8,0,0,1],[0,0,0,0,0,5,0,0,9],
     [6,4,0,0,0,0,0,7,3],[1,0,0,4,0,0,0,0,0],
     [3,0,0,7,0,0,0,6,0],[0,0,0,1,4,6,0,0,2],
     [2,0,6,0,0,0,0,1,0]]
```

Les 9 carrés de taille 3×3 sont numérotés du haut à gauche jusqu'en bas à droite. Ainsi, sur cette grille, le carré 0, en haut et à gauche, contient les chiffres 6, 4 et 7 ; le carré 1, en haut et au milieu, contient les chiffres 9, 2, 1 et 8 ; le carré 8, en bas et à droite, contient les chiffres 6, 2 et 1.

On rappelle que les lignes du Sudoku sont alors les éléments de `L` accessibles par `L[0]`, ..., `L[8]`. L'élément de la case (i, j) est accessible par `L[i][j]`.

Remarque : on fera bien attention, dans l'ensemble de ce sujet, aux indices des tableaux. Les lignes, ainsi que les colonnes, sont indicées de 0 à 8.

Question 1.1. Résultats préliminaires

(a) Montrer que si une grille de Sudoku est complète, alors pour chacune des lignes, chacune des colonnes et chacun des carrés de taille 3×3, la somme des chiffres fait 45. La réciproque est-elle vraie ?

(b) Écrire une fonction `ligne_complete(L,i)` qui prend une liste Sudoku `L` et un entier i entre 0 et 8, et renvoie `True` si la ligne i du Sudoku `L` vérifie les conditions de remplissage d'un Sudoku, et `False` sinon.

On définit de même (on ne demande pas de les écrire) les fonctions `colonne_complete(L,i)` pour la colonne i et `carre_complet(L,i)` pour le carré i.

(c) Écrire une fonction `complet(L)` qui prend une liste Sudoku `L` comme argument, et qui renvoie `True` si la grille est complète, `False` sinon.

Question 1.2. Fonctions annexes

(a) Compléter la fonction suivante `ligne(L,i)`, qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent sur la ligne d'indice i. *Ainsi, avec la grille donnée dans l'énoncé, on doit obtenir le résultat `[6,3,5]` si on entre `ligne(L,0)`.*

```
def ligne(L ,i ):
    chiffre = []
    for j in ....:
        if ....:
            chiffre.append(L[i][j])
    return chiffre
```

On définit alors, de la même manière, la fonction `colonne(L,j)` qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent dans la colonne j (on ne demande pas d'écrire son code).

(b) On se donne une case (i,j), avec $(i,j) \in [\![0;8]\!]^2$. Montrer que la case en haut à gauche du carré 3×3 auquel appartient la case (i,j) a pour coordonnées $(3\times\lfloor i/3\rfloor, 3\times\lfloor j/3\rfloor)$ où $\lfloor x\rfloor$ représente la partie entière de x.

(c) Compléter alors la fonction `carre(L,i,j)`, qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent dans le carré 3×3 auquel appartient la case (i,j). *On rappelle que si `x` et `y` sont des entiers, `x//y` renvoie le quotient de la division euclidienne de `x` par `y`. Ainsi, avec la grille donnée dans l'énoncé, `carre(L,4,6)` doit renvoyer `[9, 7, 3]` et `carre(L,4,5)` doit renvoyer `[5, 4]`.*

```
def carre(L, i, j) :
    icoin = 3*(i//3)
    jcoin = 3*(j//3)
    chiffre = []
    for i in range(....):
        for j in range(....) :
            if ....:
                chiffre.append(L[i][j])
    return chiffre
```

(d) Déduire des questions précédentes une fonction `conflit(L,i,j)` renvoyant la liste des chiffres que l'on ne peut pas écrire en case (i,j) sans contredire les règles du jeu. *La liste renvoyée peut très bien comporter des redondances. On ne prendra pas en compte la valeur de `L[i][j]`.*

(e) Compléter enfin la fonction `chiffres_ok(L,i,j)` qui renvoie la liste des chiffres que l'on peut écrire en case (i,j). *Par exemple, avec la grille initiale, `chiffres_ok(L,4,2)` renvoie `[2, 5, 8, 9]`.*

```
def chiffres_ok(L, i, j):
    ok = []
    conflit = conflit(L, i, j)
    for k in ....:
        if ....:
            ok.append(k)
    return ok
```

On pourra, dans la suite du sujet, utiliser les fonctions annexes définies précédemment.

Question 1.3. Algorithme naïf Naïvement, on commence par compléter les cases n'ayant qu'une seule possibilité. Nous prendrons dans la suite comme Sudoku :

```
M = [[2, 0, 0, 0, 9, 0, 3, 0, 0], [0, 1, 9, 0, 8, 0, 0, 7, 4],
     [0, 0, 8, 4, 0, 0, 6, 2, 0], [5, 9, 0, 6, 2, 1, 0, 0, 0],
     [0, 2, 7, 0, 0, 0, 1, 6, 0], [0, 0, 0, 5, 7, 4, 0, 9, 3],
     [0, 8, 5, 0, 0, 9, 7, 0, 0], [9, 3, 0, 0, 5, 0, 8, 4, 0],
     [0, 0, 2, 0, 6, 0, 0, 0, 1]]
```

(a) A partir des fonctions annexes, écrire une fonction `nb_possible(L,i,j)`, indiquant le nombre de chiffres possibles à la case (i,j).

On souhaite disposer de la fonction `un_tour(L)` qui parcourt l'ensemble des cases du Sudoku et qui complète les cases dans le cas où il n'y a qu'un chiffre possible, et renvoie `True` s'il y a eu un changement, et `False` sinon. La liste `L` est alors modifiée par effet de bords. Par exemple, en partant de la grille initiale `M` :

```
>>> un_tour(M)
True
>>> M
[[2, 0, 0, 0, 9, 0, 3, 0, 0], [0, 1, 9, 0, 8, 0, 5, 7, 4],
 [0, 0, 8, 4, 0, 0, 6, 2, 9], [5, 9, 0, 6, 2, 1, 4, 8, 7],
 [0, 2, 7, 0, 3, 8, 1, 6, 5], [0, 6, 1, 5, 7, 4, 2, 9, 3],
 [0, 8, 5, 0, 0, 9, 7, 3, 0], [9, 3, 6, 0, 5, 0, 8, 4, 2],
 [0, 0, 2, 0, 6, 0, 9, 5, 1]]
```

On propose la fonction suivante :

```
def un_tour(L):
    changement = False
    for i in range(1, 9):
        for j in range (1, 9):
            if (L[i][j] = 0):
                if (nb_possible(L, i, j) = 1):
                    L[i][j] = chiffres_ok(L, i, j)[1]
    return changement
```

(b) Recopier ce code en en corrigeant les erreurs.

(c) Écrire une fonction `complete(L)` qui exécute la fonction `un_tour` tant qu'elle modifie la liste, et renvoie `True` si la grille est complétée, et `False` sinon.

Question 1.4. Backtracking La deuxième idée est de résoudre la grille par « Backtracking » ou « retour-arrière ». L'objectif est d'essayer de compléter la grille de Sudoku en testant les combinaisons, en commençant par la première case, et jusqu'à la dernière. Si on obtient un conflit avec les règles, on est obligé de revenir en arrière. On va compléter la grille en utilisant l'ordre lexicographique, c'est-à-dire les cases $(0,0), \ldots, (0,8)$ puis $(1,0), (1,1), \ldots, (1,8), (2,0), \ldots$.

(a) Écrire une fonction `case_suivante(pos)` qui prend une liste `pos` du couple des coordonnées de la case, et renvoie la liste du couple d'indices de la case suivante en utilisant l'ordre lexicographique, et qui renvoie `[9,0]` si `pos=[8,8]`. *Par exemple, `case_suivante([1,3]` renvoie `[1,4]`.*

La fonction principale va avoir la structure suivante :

```
def solution_sudoku(L):
    return backtracking(L, [0, 0])
```

où `backtracking(L,pos)` est une fonction récursive qui doit renvoyer `True` s'il est possible de compléter la grille à partir des hypothèses faites sur les cases qui précèdent la case `pos`, et `False` dans le cas contraire. Ainsi :

— Si `pos` est la liste `[9, 0]`, la grille est complétée, et on renvoie `True` (cas d'arrêt).
— Si la case est déjà remplie (donnée initiale du Sudoku), on passe à la case suivante via un appel récursif.
— Sinon, on affecte un des chiffres possibles à la case, et on passe à la case suivante par un appel récursif.

(b) Compléter le squelette de la fonction `backtracking(L,pos)` selon les règles précédentes.

```
def backtracking(L, pos):
    """
    pos est une liste désignant une case du sudoku,
    [0 ,0] pour le coin en haut à gauche.
    """
    if pos == [9, 0]:
        ....
    i, j = pos[0], pos[1]
    if L[i][j] != 0:
        return ....
    for k in .... :
        L[i][j] = ....
        if .... :
            return ....
    L[i][j] = ....
    return ....
```

(c) On suppose qu'au départ, il y a p cases déjà remplies. Montrer qu'au maximum, la fonction `backtracking` est appelée 9^{81-p} fois.

(d) Que renvoie la fonction `solution_sudoku(L)` si le sudoku `L` admet plusieurs solutions ? Et si `L` est le sudoku rempli de 0 ?

(e) Dans l'algorithme précédent, on parcourt l'ensemble des cas dans l'ordre lexicographique. Comment améliorer celui-ci pour limiter le nombre d'appels à la variable `pos` ?