Exercice 40

Exercice 40 (Sudoku). Le *Sudoku* est un jeu de grille popularisé en 2005, qui consiste dans sa version classique à compléter une grille 9×9 partiellement remplie, en respectant les règles suivantes :

(1) sur chacune des 9 lignes, on trouve les chiffres de 1 à 9 (chacun apparaissant exactement une fois);
(2) sur chacune des 9 colonnes, on trouve les chiffres de 1 à 9 (chacun apparaissant exactement une fois);
(3) si l'on découpe la grille en 9 sous-grilles disjointes de taille 3×3, alors dans chacune de ces 9 sous-grilles, on trouve les chiffres de 1 à 9 (chacun apparaissant exactement une fois).

En général, une grille de Sudoku est conçue pour qu'il n'existe qu'une seule solution (c'est-à-dire qu'une seule façon de la compléter en respectant les règles ci-dessus). La grille ci-dessous est un exemple d'une telle grille : les points représentent les cases à compléter, et il n'y a qu'une seule façon de le faire en respectant les 3 règles que nous venons d'énoncer. Cela n'a rien d'évident a priori, mais lorsque l'on complète peu à peu la grille par des enchaînements de déductions logiques, on constate effectivement que l'on arrive à une grille complète (avec toutes les case remplies) qui est entièrement déterminée par les valeurs connues au départ.

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

(1) On choisit de représenter une grille de Sudoku (partiellement ou complètement remplie) par une liste `G` contenant 9 listes de 9 valeurs (chacune de ces 9 listes codant une ligne de la grille). Ainsi, la valeur `G[i][j]` contient, pour $0 \leqslant i, j \leqslant 8$, soit la valeur connue (un entier entre 1 et 9) présente sur la ligne `i` et la colonne `j` de la grille (la numérotation commençant à 0), soit, si la case n'est pas encore remplie sur la grille, la valeur `None`. Avec cette représentation, la grille précédente pourrait être définie par

```
G = [[2,None,6,9,None,4,None,None,None],[None,None,5,None,None,6,9,8,7], ... ]
```

(les points de suspension désignent la définition des 7 dernières listes, non reproduite ici), et l'on aurait par exemple `G[0][0]=2`, `G[1][1]=None` et `G[1][2]=5`. Pour plus de commodité, on souhaiterait pouvoir définir une grille (au format que nous venons d'évoquer : une liste de 9 listes) à partir d'une seule chaîne de 81 caractères, en codant ligne après ligne les valeurs connues par des chiffres et les valeurs inconnues par des points. Ainsi, la grille précédente pourrait être définie à partie de la chaîne de caractères

```
s = '2.69.4.....5..69878....5..23..6..25.65.....73.91..3..45..1....81634..7.....5.74.6'
```

Écrire une fonction Python `lit_grille()`, qui prend en entrée une chaîne de 81 caractères, formée uniquement de chiffres 1 à 9 et de points (comme dans l'exemple ci-dessus) et renvoie une représentation de la grille associée sous la forme d'un liste de 9 listes de 9 entiers (le point étant codé par la valeur `None`).

(2) Inversement, on souhaiterait pouvoir afficher une grille sous une forme lisible. Écrire une fonction `affiche_grille()`, qui prend en entrée une grille de Sudoku représentée par une liste de 9 listes comme nous l'avons vu plus haut, et affiche cette grille ligne par ligne avec des chiffres et des points séparés par des espaces. Voici un exemple de résultat attendu après utilisation des fonctions `lit_grille()` et `affiche_grille()` :

```
>>> s='2.69.4.....5..69878....5..23..6..25.65.....73.91..3..45..1....81634..7.....5.74.6'
>>> G=lit_grille(s) # convertit la chaîne s en une grille représentée par listes
>>> affiche_grille(G) # affiche la grille G
2 . 6 9 . 4 . . .
. . 5 . . 6 9 8 7
8 . . . . 5 . . 2
3 . . 6 . . 2 5 .
6 5 . . . . . 7 3
. 9 1 . . 3 . . 4
5 . . 1 . . . . 8
1 6 3 4 . . 7 . .
. . . 5 . 7 4 . 6
```

(3) Pour résoudre une grille de Sudoku, on est amené à chercher, pour une case donnée (non remplie), l'ensemble des chiffres compatibles avec le respect des 3 règles énoncés plus haut concernant la ligne, la colonne, et la sous-grille 3×3 contenant cette case. Par exemple, la case `G[1][0]` coloriée en vert ci-dessous

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

- ne peut contenir les chiffres 5,6,7,8,9 car ceux-ci sont déjà présents sur la ligne d'indice 1;
- ne peut contenir les chiffres 1,2,3,5,6,8 car ceux-ci sont déjà présents sur la colonne d'indice 0;
- ne peut contenir les chiffres 2,5,6,8 car ceux-ci sont déjà présents sur la sous-grille haut-gauche.

On en déduit que les possibilités pour cette case sont des éléments de l'ensemble

$$\{1,2,3,4,5,6,7,8,9\} \setminus \big(\{5,6,7,8,9\} \cup \{1,2,3,5,6,8\} \cup \{2,5,6,8\}\big) = \{4\}.$$

Comme cet ensemble n'a qu'un élément, la seule possibilité est `G[1][0]=4`.

Compléter la fonction Python `possibles()` ci-dessous en remplaçant les pointillés. Cette fonction prend en entrée une grille de Sudoku G ainsi que deux indices i, j tels que $0 \leqslant i, j \leqslant 8$, et renvoie l'ensemble des chiffres possibles pour la case (i, j) de G, obtenu en retirant de l'ensemble {1,2,3,4,5,6,7,8,9} les valeurs présentes sur la ligne i de G, la colonne j de G, et la sous-grille 3×3 de G contenant la case (i, j).

```
def possibles(G,i,j):
    # valeurs présentes sur la ligne i
    L = set(G[i][k] for k in range(9))
    # valeurs présentes sur la colonne j
    C = ..............
    # valeurs présentes dans la sous-grille contenant la case (i,j)
    i0,j0 = ..............
    S = set(G[i0+k][j0+l] for k in range(3) for l in range(3))
    return {1,2,3,4,5,6,7,8,9} - ..............
```

Tester ensuite cette fonction en vérifiant que le résultat obtenu par l'appel `possibles(G,1,0))` est conforme à ce que l'on a prédit plus haut.

(4) Nous allons maintenant écrire une fonction `résoudre(G)` qui va permettre de résoudre une grille de Sudoku, avec une simple stratégie récursive. Cette fonction prend une grille (potentiellement incomplète) `G`, la modifie, et

- renvoie `True` si cette grille admet une solution (c'est-à-dire, peut-être complétée en respectant les règles 1,2,3); dans ce cas, la grille `G` contient, après appel de la fonction, la solution trouvée;
- renvoie `False` s'il n'y a aucune solution.

Écrire en Python la fonction `résoudre(G)` à partir du pseudo-code suivant :

fonction résoudre(G)
| // renvoie `vrai` si la grille G admet une solution, `faux` sinon
| // la grille G est modifiée, et contient la solution si la fonction renvoie `vrai`
| pour $i = 0, 1, \ldots 8$
| | pour $j = 0, 1, \ldots 8$
| | | si la case (i, j) de G n'est pas déjà remplie
| | | | // on essaie de trouver une solution en remplissant $G(i, j)$
| | | | pour toute valeur x dans possibles(G, i, j)
| | | | | remplir la case (i, j) de G avec la valeur x
| | | | | si résoudre(G)
| | | | | | retourner `vrai`
| | | | // on n'a pas trouvé de solution en remplissant $G(i, j)$
| | | | vider la case (i, j) de G
| | | | retourner `faux`
| retourner `vrai` // car toutes les cases de G sont déjà remplies

Tester ensuite le code proposé avec

```
s = '2.69.4.....5..69878....5..23..6..25.65.....73.91..3..45..1....81634..7.....5.74.6'
G = lit_grille(s)
affiche_grille(G) # grille de départ (incomplète)
print(résoudre(G)) # doit afficher True
affiche_grille(G) # affichage de la solution
```

Tester ensuite l'algorithme sur des grilles de votre choix (trouvées sur internet par exemple). Pouvez-vous trouver une grille que la fonction `résoudre()` met plus de 10 secondes à résoudre ? (donner la "pire" grille trouvée et le temps de calcul associé)

(5*) L'algorithme que nous avons utilisé pour la fonction `résoudre()` est assez élémentaire, mais permet néanmoins de résoudre quasi instantanément la plupart des grilles de Sudoku. Pour la grille définie par

```
s = '.......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........8.6...'
```

cependant, un tel algorithme n'est pas assez efficace pour trouver la solution en un temps raisonnable. Proposer une version améliorée de la fonction `résoudre()` qui permet de trouver la solution en moins d'une minute. On pourra, par exemple, choisir plus soigneusement la case (i, j) que l'on complète, en privilégiant l'une des cases ayant le moins de possibilités.