TNSI	Langages & programmation Récursivité	**TP – Sudoku**

L'objectif de ce TP est de réaliser un solveur de grille de Sudoku. Il est fortement inspiré du travail de Jules Svartz du Lycée Masséna à Nice.

Une grille de sudoku est un tableau de 9 × 9 cases. Les cases contiennent les chiffres de 1 à 9 et au début seules quelques cases sont remplies (les autres sont vides).
Si vous n'êtes pas à l'aise avec les règles du sudoku, voir https://sudokus.fr/regles-sudoku/.
La grille de sudoku sera représentée en machine par un tableau `grille` à deux dimensions : chaque élément du tableau représente une ligne et contient un tableau dans lequel chaque case correspond à une colonne.
Les cases contiennent un entier (`int`) de 1 à 9 ou bien la valeur `0` si la case est vide.

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

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

⇒ Le tableau `grille` est de taille 9 ;
⇒ les « lignes » du sudoku sont les éléments de `grille`, accessibles par `grille[0]` (la ligne du haut) jusqu'à `grille[8]`. Ce sont des tableaux de taille 9.
⇒ L'élément en case (i, j) (ligne i, colonne j, pour $0 \leq i, j \leq 8$, numérotées depuis le coin en haut à gauche) est accessible par `grille[i][j]`. Par exemple, le 7 en gras et rouge en haut du sudoku est l'élément `grille[1][5]`.
⇒ Les cases non remplies sont associées au chiffre 0.

Le but du programme est de compléter la grille de sudoku en remplissant chaque case vide par des chiffres de 1 à 9 de sorte que dans chaque ligne, chaque colonne, et chaque bloc de taille 3 × 3, chaque chiffre n'apparaisse qu'une et une seule fois. Certaines cases sont naturellement déjà remplies, et un Sudoku bien écrit ne possède normalement qu'une seule solution.
On va décrire une solution efficace au remplissage d'une grille de sudoku par backtracking[1].
La technique du backtracking consiste simplement à essayer de remplir le sudoku en commençant par la première case jusqu'à la dernière. Si l'on découvre un conflit avec les règles, on est obligé de revenir en arrière. Le retour en arrière est considérablement simplifié par l'usage de la récursivité.

I – Détection de conflits

Question 1 :
Écrire une fonction `chiffres_ligne(grille,i)` renvoyant la liste des nombres de 1 à 9 qui apparaissent sur la ligne d'indice i sous la forme d'un tableau. Par exemple, avec la grille initiale `chiffres_ligne(grille,0)` doit donner `[9, 2, 6, 5]`.

[1] Le backtracking est une technique de programmation permettant au programme de revenir en arrière quand les choix effectués jusque-là se révèlent inadaptés.

Question 2 :
Faire de même avec `chiffres_colonne(grille,j)`.
`chiffres_colonne(grille,3)` doit donner `[2,9,4]`.

Question 3 :
Écrire maintenant une fonction `chiffres_bloc(grille,i,j)` fournissant la liste des nombres de 1 à 9 apparaissant dans le bloc de taille 3 × 3 auquel appartient la case (i, j). Exemple : `chiffres_bloc(grille,4,7)` doit donner `[9, 4]` et `chiffres_bloc(grille,4,5)` doit donner `[]`.

Indication : i%3 fournit le résultat du reste dans la division euclidienne de i par 3.

Question 4 :
Déduire des questions précédentes une fonction `chiffres_conflit(grille,i,j)` retournant la liste des chiffres qu'on ne peut pas écrire en case (i, j) sans contredire les règles du jeu. On ne se préoccupera pas du fait que `grille[i][j]` puisse être dans la liste s'il est non nul, on n'appliquera cette fonction dans la suite que si `grille[i][j]` est nul. Ça n'a aucune importance si certains chiffres apparaissent plusieurs fois. Exemple : `chiffres_conflit(grille,0,0)` doit donner `[9, 2, 6, 5, 3, 6, 9, 3, 2, 7]`.

II - Passage à la case suivante

On essaye de remplir la grille par ligne croissante (de i = 0 à i = 8), puis par colonne croissante (de j = 0 à j = 8). En clair, on va d'abord essayer de mettre un chiffre en case (0, 0), puis en case (0, 1), etc... Si on a pu mettre un chiffre en case (0, 8), on passe à la case (1, 0), etc...

Question 5 :
Écrire une fonction `case_suivante(i,j)` permettant d'obtenir un couple d'indices indiquant les coordonnées de la case suivante. On renverra sans se poser de question `(9, 0)` lorsque la fonction est appelée avec pour argument `(8, 8)`.

III - La fonction principale

On va maintenant écrire une fonction permettant de résoudre un Sudoku. Voici un squelette d'une fonction prenant en entrée un tableau `grille` représentant un Sudoku.

```
def solution_sudoku(grille:list):
    def completer_grille(i,j):
        [...]
    return completer_grille(0,0)
```

La fonction récursive `completer_grille(i,j)` doit renvoyer `True` si l'on a réussi à compléter toute la grille à partir des hypothèses faites dans les cases précédant (i, j) (dans l'ordre de la question précédente) et `False` dans le cas contraire.

Le principe est le suivant :

- Le cas de base est lorsqu'on a réussi à remplir la grille : on a i = 9 et j = 0. Dans ce cas on renvoie `True`.
- Si la case `grille[i][j]` est déjà remplie (c'était une des données du Sudoku), il n'y a rien à faire, on passe à la case suivante.
- Sinon, on calcule les chiffres que l'on ne peut pas mettre en case (i, j), et on essaie successivement tous les autres : on écrit un chiffre en case (i, j) et on passe en case suivante par appel récursif. Il est nécessaire de récupérer le booléen associé à un tel appel : s'il est `True` on a réussi à remplir la grille à partir des suppositions sur les cases précédentes, s'il est `False` c'est que l'essai n'est pas concluant et qu'il faut essayer un autre chiffre.

La fonction principale se contente de l'appel `completer_grille(0,0)` : il faut essayer de tout remplir à partir du début !

Question 6 :
Écrire la fonction `solution_sudoku(grille)`.
Avec la grille d'exemple, la fonction doit renvoyer `[[8, 9, 1, 2, 3, 4, 6, 7, 5], [3, 2, 5, 6, 8, 7, 4, 1, 9], [4, 7, 6, 9, 1, 5, 3, 2, 8], [9, 1, 4, 7, 5, 2, 8, 6, 3], [2, 5, 7, 3, 6, 8, 1, 9, 4], [6, 8, 3, 1, 4, 9, 7, 5, 2], [1, 4, 8, 5, 2, 6, 9, 3, 7], [7, 3, 2, 4, 9, 1, 5, 8, 6], [5, 6, 9, 8, 7, 3, 2, 4, 1]]`

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

Solution trouvée par l'algorithme

Question 7 :
Quelle est la classe de complexité de cet algorithme (fonction `completer_grille`) ?

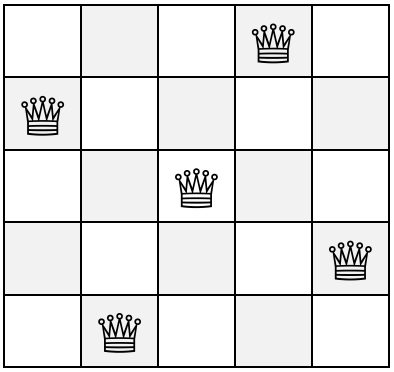
Question 8 :
Imaginons une variante du sudoku pour des grilles de taille n × n. La taille de la pile de récursivité est par défaut de 1000 en Python. Quelle taille maximale de n peut-on envisager avec un algorithme similaire sans augmenter la taille de la pile ? Justifier.

Question 9 :
Que donne votre fonction si la liste passée en argument est erronée, c'est à dire que le Sudoku n'a pas de solution, ou au contraire plusieurs ?

Question 10 :
Un autre problème pour ceux qui ont terminé : quelles sont les solutions au problème des 8 dames ? Le principe est de positionner 8 dames sur un échiquier de sorte que deux d'entre elles ne soient jamais en prise (i.e sur la même ligne, la même colonne, ou la même diagonale). Vous écrirez une fonction qui marche sur le même principe, et une fonction pour afficher les solutions à l'écran. La fonction doit renvoyer toutes les possibilités.

Aide : Vous pouvez regarder la vidéo https://www.youtube.com/watch?v=R8bM6pxlrLY qui explique en danse le principe de l'algorithme de backtracking.

Davantage d'aide :
On sait déjà que les dames ne peuvent pas être sur la même ligne. Comme il y a n dames et n lignes, alors chaque dame occupe une ligne. Plutôt que de stocker pour chaque dame sa ligne et sa colonne, on peut donc enregistrer pour chaque dame uniquement la colonne où elle se trouve. Ainsi on aura un tableau d'entiers `dames` où le premier élément contiendra la colonne de la dame située sur la ligne 0, le deuxième la colonne de la dame située sur la ligne 1, ... et le dernier élément donne la colonne de la dernière dame située sur la ligne n-1.
S'il n'y a pas de dame dans la ligne considérée, on stocke -1 comme numéro de colonne.
Le principe de l'algorithme est à chaque étape d'essayer de fixer la colonne de la $k^{ième}$ dame. Si la colonne choisie est possible, on essaye récursivement de remplir la ligne k+1. L'algorithme s'arrête quand k = n-1 (toutes les dames sont placées).
Il faudra écrire une fonction `est_menace(dames, ligne_cible, colonne_cible)` renvoyant `True` si la case cible est menacée par une des autres dames et une fonction récursive `config_n_dames(n, k, dames)`.



`dames = [3,0,2,4,1]`