TP : Algorithmes de Backtracking et résolution de Sudoku

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

```
L=[[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]]
```

FIGURE 1: Une grille de Sudoku non encore remplie et sa représentation en Python

On se donne une grille de Sudoku sous la forme d'une liste de taille 9×9. Le but est d'y écrire des chiffres de $[\![1, 9]\!]$ de sorte que dans chaque ligne, chaque colonne, et chaque bloc de taille 3×3, chaque chiffre apparaisse une et une seule fois. Certaines cases sont naturellement déja 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. À partir de maintenant, la grille de sudoku est représentée en Python [1] par une liste de listes, comme dans la figure précédente :

- la liste `L` est de taille 9.
- les « lignes » du sudoku sont les éléments de `L`, accessibles par `L[0]` (la ligne du haut) jusqu'à `L[8]`. Ce sont des listes.
- 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 `L[i][j]`. Par exemple, le 9 en haut du sudoku est l'élément `L[0][1]`.
- les cases non remplies sont associées au chiffre 0.

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(L,i)` renvoyant la liste des nombres de 1 à 9 qui apparaîssent sur la ligne d'indice i. Par exemple, avec la grille initiale :

```
>>> chiffres_ligne(L,0)
[9, 2, 6, 5]
```

Question 2. Faire de même avec `chiffres_colonne(L,j)`.

Question 3. Écrire maintenant une fonction `chiffres_bloc(L,i,j)` fournissant la liste des nombres de 1 à 9 apparaîssant dans le bloc de taille 3×3 auquel appartient la case (i, j). *Indication :* `i%3` fournit le résultat du reste dans la division euclidienne de i par 3.

```
>>> chiffres_bloc(L,4,7)
[9, 4]
>>> chiffres_bloc(L,4,5)
[]
```

Question 4. Déduire des questions précédentes une fonction `chiffres_conflit(L,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 `L[i][j]` puisse être dans la liste s'il est non nul, on n'appliquera cette fonction dans la suite que si `L[i][j]` est nul. Ça n'a aucune importance si certains chiffres apparaîssent plusieurs fois.

```
>>> chiffres_conflit(L,0,0)
[9, 2, 6, 5, 3, 6, 9, 3, 2, 7]
```

1. Voir mon site web pour une liste d'une cinquantaine de Sudoku en Python.

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 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 une liste `L` représentant un Sudoku.

```
def solution_sudoku(L):
    def aux(i,j):
        [...]
    return aux(0,0)
```

La fonction récursive `aux(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) (pour 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 `L[i][j]` est déja 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.

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

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

FIGURE 2: La grille résolue, par backtracking.

Question 6. C'est la question importante : écrire la fonction `solution_sudoku(L)`.

Question 7. Imaginons une variante du sudoku pour des grilles de taille $n \times n$. 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 8. 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 ? Calculez le plus petit Sudoku pour l'ordre lexicographique.

Question 9. 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.