L'objectif de ce TP est de mettre en œuvre un programme permettant de résoudre automatiquement une grille de Sudoku.

Document 1 : Règles du jeu

Le Sudoku, actualisé par Wayne Gould est un jeu subtil, de logique, de mémoire et d'observation. Ce jeu s'adresse à tous, petits et grands. Les règles sont très simples mais remplir sans erreur une grille de Sudoku peut être, pour certaines grilles, très difficile.

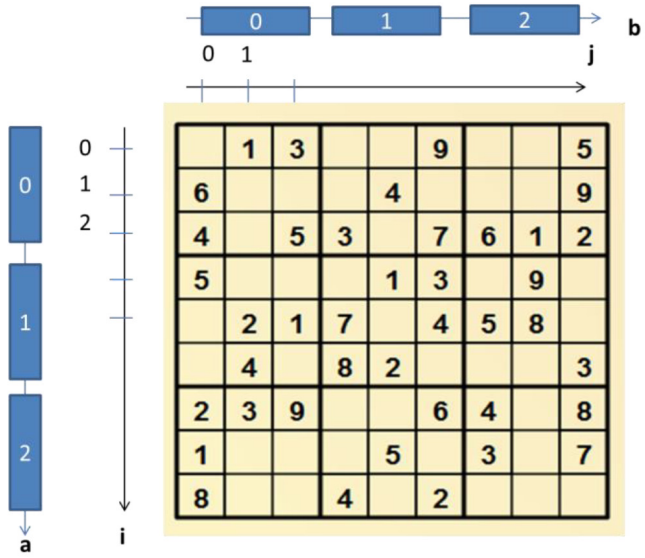
Principe : Il s'agit de compléter une grille de 9 cases sur 9 cases, subdivisée en 9 carrés appelés régions, avec des chiffres allant de 1 à 9.

- Chaque chiffre ne doit apparaitre qu'une seule fois dans chaque ligne,
- Chaque chiffre ne doit apparaitre qu'une seule fois dans chaque colonne,
- Chaque chiffre ne doit apparaitre qu'une seule fois dans chaque région.

Quelques chiffres sont déjà placés correctement dans la grille. Le joueur doit placer les autres chiffres. Plusieurs niveaux de difficultés sont proposés. Une grille doit toujours posséder une solution logique pour être proposée au public.

Document 3 : Exemple de grille de Sudoku et son paramétrage

Afin de repérer une case dans la grille on utilise ses coordonnées i et j définies ci-dessous et en commençant par 0 sous Python. Une région est, elle, définie par ses coordonnées a et b également définies ci-dessous.



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

1. Quel(s) chiffre(s) peu(ven)t convenir pour la case de coordonnées $(0,0)$ de la grille du document 3 ?

Dans ce problème une grille de Sudoku sera décrite par une matrice de type « list ». Les cases vides seront initialement remplies par des 0.

2. Écrire la ligne de commande permettant de définir en Python la matrice `grille` qui correspond à la grille du document 3 dans son état initial.
3. Que doit alors renvoyer la commande `grille[1][3]` ?

Pour pouvoir jouer, on réalise la fonction suivante :

```
def cv(M):
    """Que peut bien faire cette fonction?"""
    vide=[]
    for i in range(9):
        for j in range(9):
            if M[i][j]==0:
                vide=vide+[[i,j]]
    return vide
```

4. Quel est le rôle de la fonction **cv**`(M)` ?

Après avoir défini la matrice `grille` dans la question 2, on écrit la commande `liste_cv=cv(grille)`.

5. Donner le contenu de `liste_cv`.

Il faut maintenant apprendre à notre programme les règles du jeu en détails.

6. Écrire une fonction **test_ligne**`(chiffre,i,j,M)` qui renvoie
   - `True` si mettre `chiffre` dans la case de coordonnées (i, j) dans la matrice `M` respecte la condition d'unicité sur la ligne i,
   - `False` sinon.
7. Écrire une fonction **test_colonne**`(chiffre,i,j,M)` qui renvoie
   - `True` si mettre `chiffre` dans la case de coordonnées (i, j) dans la matrice `M` respecte la condition d'unicité sur la colonne j,
   - `False` sinon.
8. En utilisant les propriétés de la division euclidienne, écrire une fonction **region**`(i,j)` qui renvoie la liste $[a, b]$ donnant les coordonnées de la région dans laquelle se trouve la case de coordonnées (i, j)
9. À partir de la fonction précédente, écrire une fonction **test_region**`(chiffre,i,j,M)` qui renvoie
   - `True` si mettre `chiffre` dans la case de coordonnées (i, j) dans la matrice `M` respecte la condition d'unicité sur la région,
   - `False` sinon.
10. A partir des fonctions précédentes, mettre en œuvre une fonction `test(chiffre,i,j,M)` qui renvoie
    - `True` si mettre `chiffre` dans la case de coordonnées (i, j) dans la matrice `M` respecte toutes les conditions d'unicité
    - `False` sinon.

Pour résoudre une grille de Sudoku, on peut employer diverses méthodes. L'une des plus simples à coder, même si ce n'est pas la plus rapide, est la méthode du « backtracking ».

Document 4 : Principe du backtracking appliqué au Sudoku

Les cases vides sont d'abord identifiées. On teste alors la première case vide en lui attribuant la valeur 1. Si cette valeur respecte les conditions d'unicité, alors on teste la case vide suivante. Sinon on incrémente la valeur de la case vide testée de 1 en 1 jusqu'à trouver une valeur qui convienne avant de passer à la case vide suivante. Si aucune valeur ne convient pour une case testée, cela signifie qu'une valeur attribuée précédemment n'est finalement pas la bonne. Il est alors nécessaire de revenir en arrière et d'augmenter le chiffre de la case vide précédente. Ce retour en arrière a lieu autant de fois qu'il subsiste une incompatibilité concernant le choix d'un chiffre.
Les algorithmes mettant en œuvre une méthode de backtracking sont généralement codés de manière récursive. Dans le cas de la grille de Sudoku :

- On identifie les case vides.
- S'il n'y a pas de case vide, la grille est remplie. On renvoie alors directement la grille.
- Sinon, on examine la première case vide : On cherche la première valeur possible (de 1 à 9) pour cette case. (alors elle n'est plus vide, ou plus exactement pour nous ici, elle ne vaut plus 0)
- Si, en appliquant le même procédé à la grille ainsi partiellement (et provisoirement) complétée, aucune incompatibilité n'a été décelé (on a de la chance on a toujours choisi la bonne valeur) alors le procédé se répète tant qu'il y a des cases vides,
- Si, au cours du remplissage, une incompatibilité apparait, on revient en arrière en « vidant » la dernière case remplie et en choisissant une valeur plus élevée compatible. S'il n'y en a pas on retourne en arrière autant de fois que nécessaire.

11. Écrire la fonction `sudoku(M)` qui retourne la grille M remplie ou `False` si la grille n'a pas de solution.
12. Résoudre la grille donnée en document 3.