Python – Sudokus

Le Sudoku

Le Sudoku est un jeu inventé par l'américain Howard Gams en 1979. Il consiste à remplir une grille carrée de 9 cases de côté, découpée en 9 carrés de 3 cases par côté. Chaque ligne, chaque colonne et chacun des sous-carrés doit contenir exactement une fois chacun des chiffres de 1 à 9. La grille est proposée avec des trous et il faut la remplir en respectant les contraintes. Pour chaque grille proposée, il n'y a qu'une seule solution possible.
La résolution de ces grilles peut être plus ou moins facile, nécessitant parfois de "tester" une valeur et éventuellement de revenir en arrière si ce choix amène à une contradiction. C'est le cas de la grille ci-contre qui est "diabolique".

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

Valeurs possibles

La méthode la plus simple pour résoudre une grille de Sudoku consiste à regarder pour chaque case le nombre de valeurs encore possible. S'il ne reste qu'une seule valeur possible, alors on place directement cette valeur dans la case.
Pour les décrire les cases, nous utiliseront les coordonnées `(i, j)` où `i` est le numéro de la ligne, en partant du haut et comptant à partir de `0` et `j` le numéro de la colonne en partant de la gauche. Dans la grille ci-dessous, la case `(2, 1)` est donc sur la ligne `2` et la colonne `1`, qui contient 7.
Dans la grille, la case `(2, 4)` a comme valeurs possibles 8 et 9. En effet, sur sa ligne, il manque 3, 6, 8 et 9, sur la colonne, il manque 2, 4, 8 et 9 et dans le sous-carré, il manque 4, 5, 8 et 9. Les seuls valeurs communes sont 8 et 9.

Exercice 1 (sur papier) : On considère la grille ci-contre.

1) Déterminer les valeurs possibles en `(2, 5)`.
2) Déterminer la seule valeur possible en `(8, 2)` et la placer.
3) Placer les valeurs des cases `(8, 4)` et `(7, 5)`.
4) Déterminer les valeurs possibles en `(2, 4)` et en `(2, 5)`.
5) Finir de résoudre la grille.

	0	1	2	3	4	5	6	7	8
0	9	6	2	1			3	7	8
1	1	8	5	6	7	3	4	2	9
2		7	4	2			5		1
3	4	9	6	5	1	7	8	3	2
4	2	1	8	9	3	6	7	4	5
5	7	5	3	8		4	1	9	6
6	5	3	1	7	6	2	9	8	4
7	8	2	7	4	5				
8	6	4		3		1			7

Représentation des grilles

Afin de représenter en Python les grilles de Sudoku, nous utiliserons des tableaux de tableaux. Chaque case contient un entier de 0 à 9, 0 correspondant à une case vide.
Ainsi la grille diabolique de l'introduction sera représentée par :

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

Pour afficher les grilles, nous pourrons utiliser la fonction suivante :

```
def afficher(grille):
    ligne_vide = "+---+---+---+"
    for i in range(9): # On parcourt les lignes
        if (i%3 == 0): # On affiche la ligne vide une ligne sur 3
            print(ligne_vide)
        ligne = ""     # On va construire la ligne
        for j in range(9): # On parcourt les cases de la ligne
            if j%3 == 0: # On met la barre une case sur 3.
                ligne += "|"
            if grille[i][j] != 0: # On voit un chiffre
                ligne += str(grille[i][j])
            else:      # On a une case vide
                ligne += "."
        ligne += "|"  # On met une barre à la fin
        print(ligne) # On affiche la ligne
    print(ligne_vide) # On affiche une ligne vide à la fin
```

```
>>> afficher(diabolique)
+---+---+---+
|37.|..9|2..|
|..6|...|..5|
|5..|27.|.4.|
+---+---+---+
|7..|6..|..2|
|...|.9.|...|
|6..|..3|..7|
+---+---+---+
|.6.|.28|..4|
|4..|...|1..|
|..9|1..|.56|
+---+---+---+
```

Importation de grilles

Pour importer les grilles, nous allons utiliser une autre syntaxe, où chaque ligne est un texte. Une fonction servira à transformer ces tableaux de textes en tableaux de tableaux d'entiers. Les cases vides pourront être représentées par des `"0"` ou tout autre symbole n'étant pas un chiffre.

```
def importer(grille_texte):
    grille = [[0]*9 for _ in range(9)] # On fait une grille vide
    for i in range(len(grille_texte)):
        for j in range(9):
            v = grille_texte[i][j]
            if v in "123456789": # On regarde si c'est un chiffre
                grille[i][j] = int(v) # On le convertit en nombre
    return grille
```

```
facile = ["902000308",
          "085600020",
          "300000060",
          "006517000",
          "210030705",
          "750804106",
          "001062084",
          "007450000",
          "640301007"]

grille_facile = importer(facile)
```

Recherche des valeurs possibles

Pour déterminer les valeurs possibles dans une case, on construit un tableau de 10 cases, toutes initialisées à 0 et à chaque fois qu'on voit une valeur dans la ligne, la colonne ou le sous-carré, on met un 1 dans la case correspondante. À la fin, on regarde les indices des cases qui sont encore à 0.

Exercice 2 : Compléter le code de la fonction `possibles(grille, i, j)` qui renvoie la liste des valeurs possibles pour la case de coordonnées `(i, j)` de `grille`. On suppose que la fonction n'est appelée que sur une case vide.

```
def possibles(grille, i, j):
    vus = [0]*10    # Tableau des valeurs vus
    for k in range(9):   # On gère la colonne
        vus[grille[k][j]] = 1
    for k in range(9):   # On gère la ligne
        vus[grille[...][...]] = ...
    for k1 in range(3):   # On parcourt les sous-carrés
        for k2 in range(3):
            i0 = 3*(i//3) + k1   # numéro de la ligne
            j0 = 3*(j//3) + k2   # numéro de la colonne
            vus[grille[...][...]] = ...
    return [k for k in range(1, 10) if vus[k] == ...]
```

```
>>> possibles(diabolique, 1, 0)
[1, 2, 8, 9]
>>> possibles(diabolique, 6, 0)
[1]
>>> possibles(diabolique, 0, 8)
[1, 8]
```

Vérification de Sudokus

Afin de vérifier qu'une grille est bien remplie et qu'elle ne comporte pas d'erreur, nous allons utiliser une fonction qui va vérifier chaque ligne, chaque colonne et chaque sous-carré.

Exercice 3 : Compléter la fonction `verifier(grille)` qui renvoie un booléen indiquant si la grille est bien remplie.

```
def verifier(grille):
    for i in range(9): # On vérifie les lignes
        vus = [0]*10
        for j in range(9):
            val = grille[i][j]
            if val > 0:
                if vus[val] == 1: # On a déjà vu la valeur
                    return ...
                vus[val] = 1
            else: # On a une case vide
                return ...
    for j in range(9): # On vérifie les colonnes
        ...
    for i0 in range(3): # On parcourt les sous-carrés
        for j0 in range(3):
            vus = [0]*10
            for i in range(3): # On parcourt les cases du sous-carré
                for j in range(3):
                    val = grille[3*i0+i][...]
                    ...
    return ...
```

```
>>> verifier(diabolique)
False
>>> verifier([[9, 6, 2, 1, 4, 5, 3, 7, 8],
              [1, 8, 5, 6, 7, 3, 4, 2, 9],
              [3, 7, 4, 2, 9, 8, 5, 6, 1],
              [4, 9, 6, 5, 1, 7, 8, 3, 2],
              [2, 1, 8, 9, 3, 6, 7, 4, 5],
              [7, 5, 3, 8, 2, 4, 1, 9, 6],
              [5, 3, 1, 7, 6, 2, 9, 8, 4],
              [8, 2, 7, 4, 5, 9, 6, 1, 3],
              [6, 4, 9, 3, 8, 1, 2, 5, 7]])
True
```

Résolution des grilles faciles

Les grilles faciles ont la particularité de pouvoir être résolues en ne remplissant que les cases n'ayant qu'une seule valeur possible. Il n'est jamais nécessaire de tester une valeur.

Exercice 4 : Compléter la fonction `placer_plus_simple(grille)` qui parcourt la grille en cherchant la première case où il n'y a qu'une seule valeur possible et place cette valeur. La fonction renvoie un booléen qui indique si une valeur a été placée ou non. Pour tester, vous pouvez décommenter la ligne avec le **`print`**.

```
def placer_plus_simple(grille):
    for i in range(9):
        for j in range(9):
            if grille[i][j] == ...: # On vérifie que la case est vide
                poss = possibles(grille, i, j)
                if len(poss) == ...:
                    #print(f"on place {poss[0]} en {(i, j)}")
                    grille[i][j] = ... # On met la première valeur
                    return ...
    return ...
```

```
>>> placer_plus_simple(diabolique)    # avec le print décommenté
on place 1 en (6, 0)
True
>>> placer_plus_simple(diabolique)    # On ne peut remplir qu'une case
False
>>> afficher(diabolique)
+---+---+---+
|37.|..9|2..|
|..6|...|..5|
|5..|27.|.4.|
+---+---+---+
|7..|6..|..2|
|...|.9.|...|
|6..|..3|..7|
+---+---+---+
|16.|.28|..4|
|4..|...|1..|
|..9|1..|.56|
+---+---+---+
```

Exercice 5 : Compléter la fonction `resoudre_simple(grille)` qui continue à remplir la grille tant qu'il reste une case qui n'a qu'une seule possibilité. À la fin la grille est affichée et la fonction renvoie le résultat de la vérification de la grille.

```
def resoudre_simple(grille):
    progres = ...
    while progres:
        progres = ...
    afficher(grille)
    return verifier(grille)
```

```
>>> resoudre_simple(grille_facile)
+---+---+---+
|962|145|378|
|185|673|429|
|374|298|561|
+---+---+---+
|496|517|832|
|218|936|745|
|753|824|196|
+---+---+---+
|531|762|984|
|827|459|613|
|649|381|257|
+---+---+---+
True
```

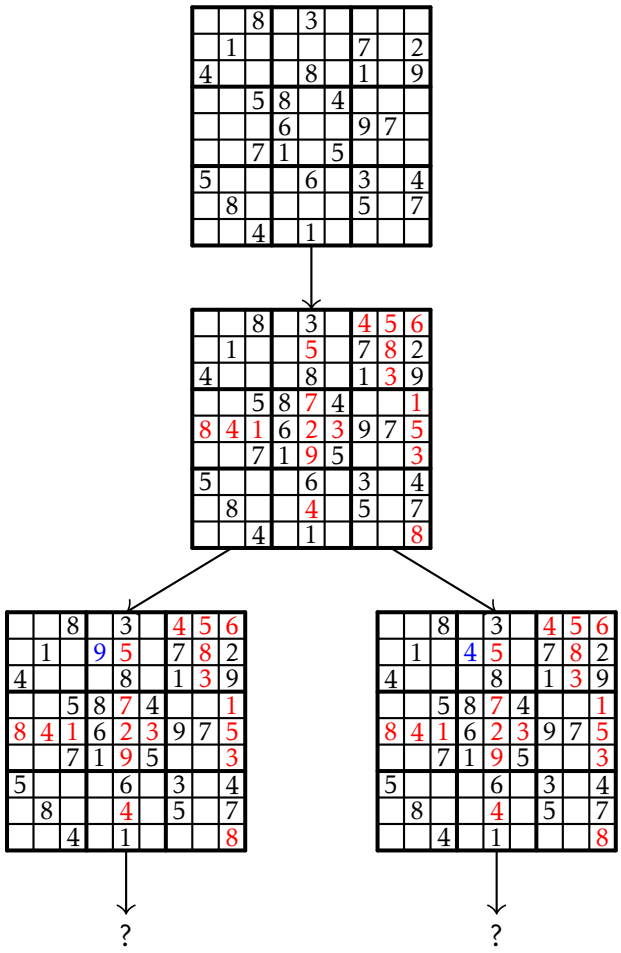
```
>>> resoudre_simple(diabolique)
+---+---+---+
|37.|..9|2..|
|..6|...|..5|
|5..|27.|.4.|
+---+---+---+
|7..|6..|..2|
|...|.9.|...|
|6..|..3|..7|
+---+---+---+
|16.|.28|..4|
|4..|...|1..|
|..9|1..|.56|
+---+---+---+
False
```

Résolution de n'importe quelle grille

Pour les grilles faciles, puisqu'il y a toujours une case qui peut être remplie de façon triviale, la résolution peut être vue comme linéaire.

Par contre pour les grilles plus complexes, il faut parfois choisir entre plusieurs valeurs pour une case. Pour chaque case, une seule des valeurs possibles mène à la solution. Toutes les autres mènent à des contradictions. Pour résoudre une grille nous allons donc utiliser l'algorithme récursif suivant :

- On commence par remplir toutes les cases faciles.
- Si on obtient une grille pleine, on a trouvé la solution.
- Si on trouve une contradiction, c'est qu'on est sur une branche qui ne mène pas à la solution.
- Si on n'a pas de contradiction, on prend une des valeurs possibles, on la met dans la grille.
- On essaie de résoudre la grille obtenue.
- Si on y arrive, on garde la solution.
- Sinon, on passe à la valeur possible suivante et on recommence.



L'exemple ci-dessus montre le début de la résolution. On part d'une grille et on place toutes valeurs faciles. On a ensuite le choix entre 4 et 9 pour la case `(1, 3)`. On essaie alors de résoudre une des deux grilles obtenues. Si on trouve la solution, c'est qu'on a fait le bon choix. Sinon on résoud l'autre grille.
On fait donc une exploration en profondeur de l'arbre de résolution.

Exercice 6 (sur papier) : On considère une des deux grilles obtenues après le choix en `(1, 3)`.

1) Déterminer les valeurs qui vont en `(1, 5)` et en `(1, 0)`.
2) Expliquer pourquoi la case `(1, 2)` montre qu'il y a une contradiction.
3) Déterminer la valeur qu'il faut donc mettre en `(1, 3)`.

	0	1	2	3	4	5	6	7	8
0			8		3		4	5	6
1		1		9	5		7	8	2
2	4				8		1	3	9
3			5	8	7	4			1
4	8	4	1	6	2	3	9	7	5
5			7	1	9	5			3
6	5				6		3		4
7		8			4		5		7
8			4		1				8

Pour pouvoir revenir en arrière lorsqu'un choix mène à une contradiction, nous allons faire une fonction qui prend la liste des coordonnées des cases qui ont été modifiées et les remet à 0.

Exercice 7 : Écrire une fonction `retirer(grille, liste)` qui met à 0 toutes les cases de `grille` dont les coordonnées sont dans `liste`.

```
>>> retirer(grille_facile, [(0, 0), (0, 2), (1, 1), (1, 2), (2, 0)])
>>> afficher(grille_facile)
+---+---+---+
|...|...|3.8|
|...|6..|.2.|
|...|...|.6.|
+---+---+---+
|..6|517|...|
|21.|.3.|7.5|
|75.|8.4|1.6|
+---+---+---+
|..1|.62|.84|
|..7|45.|...|
|64.|3.1|..7|
+---+---+---+
```

Pour résoudre les grilles, nous allons utiliser une fonction récursive. Elle fonctionnera en deux étapes :

- Tant qu'il y a des cases faciles à remplir, on les remplit. On note également les cases où il y a plusieurs choix, afin de trouver celle avec le moins de choix.
- Si lors de cette phase, il y a une case pour laquelle il n'y a aucune valeur possible, alors il y a une contradiction. On annule toutes les modifications faites et la fonction renvoie **`False`**.
- Si à la fin de cette phase, il ne reste aucune case à remplir, on a finit et on renvoie **`True`**.
- S'il reste des cases à remplir, on en prend une avec le moins de choix possibles et on les essaie les uns après les autres.
- On place la valeur testée dans la case et on fait un appel récursif avec la grille obtenue.
- Si la résolution réussit, on a fini et on renvoie **`True`**. Sinon on met la valeur suivante et on recommence.
- Si toutes les valeurs possibles mènent à une contradiction, alors on annule toutes les modifications faites au cours de cet appel et on renvoie **`False`**.

```
def resoudre_rec(grille):
    modifies = []      # Liste des nouvelles cases remplies
    coups_faciles = True
    while coups_faciles:  # On va placer les faciles
        coups_faciles = ... # On n'a rien changé ce tour de boucle
        restants = []  # On note les cases avec des choix
        for i in range(9):
            for j in range(9):
                if grille[i][j] == ...:  # Si la case est vide
                    poss = possibles(grille, i, j)
                    if len(poss) == ...:  # Il n'y a qu'une valeur
                        grille[i][j] = ... # On la place
                        coups_faciles = ... # Il faudra un nouveau tour
                        modifies.append((i, j)) # On note la case
                    elif ...: # On a plusieurs choix
                        restants.append((len(poss), i, j))
                    else: # Il n'y a aucun choix
                        retirer(grille, modifies)  # On annule les modifs
                        return ...
    if len(restants) == ...:  # On a tout placé
        return ...
    else: # On va devoir faire des choix
        restants.sort()    # On trie les cases restantes
        _, i, j = restants[0]  # On prend celui avec le moins de choix
        modifies.append((i, j))  # On note la case comme modifiée
        for v in possibles(grille, i, j): # On va tester tous les choix
            grille[i][j] = ...  # On place la valeur
            if resoudre_rec(grille):  # On essaie de résoudre la grille
                return ...  # On a réussi
        # Aucun des choix n'a marché
        retirer(grille, modifies) # On annule tout
        return ...  # Et on remonte
```

On peut alors faire une fonction qui va résoudre la grille, l'afficher et vérifier le résultat.

```
def resoudre(grille):
    res = resoudre_rec(grille)
    if not res or not verifier(grille):
        print("Il y a un problème")
    afficher(grille)
```

Vous pouvez tester cette fonction sur la grille diabolique.

```
>>> resoudre(diabolique)
```

Exercice 8 : Chercher une grille diabolique sur Internet et la résoudre grâce à la fonction ci-dessus.

Pour aller plus loin

Afin de voir les choix effectués par l'algorithme de résolution, nous allons faire de nouvelles fonctions qui construiront l'arbre des valeurs placées. La structure de cet arbre sera la suivante :

- Une valeur placée est représentée par le triplet : `(valeur, i, j)`.
- Une contradiction sera représentée par `(-1, -1, -1)`.
- La réussite de la résolution sera représentée par `(0, 0, 0)`.
- Un arbre est un tableau de deux éléments : `[faciles, choix]`.
- Le premier, `faciles`, est une liste de coups faciles, potentiellement vide, et pouvant se terminer par une contradiction ou une réussite.
- Le deuxième, `choix`, est une liste de quaduplets `(valeur, i, j, arbre)`, où `arbre` est l'arbre obtenu en plaçant `valeur` en `(i, j)`.

L'arbre ne contiendra que les branches explorées. Voici la fonction permettant d'afficher les arbres :

```
echec = (-1, -1, -1)
ok = (0, 0, 0)

def affichage_arbre(arbre, decalage="", complet=True):
    simples, choix = arbre
    fleche = "\u2794"
    if complet:
        for v, i, j in a_forces:
            if v > 0:
                print(f"{decalage}{v}{fleche}{(i, j)}")
            elif v == 0:
                print(decalage+"  Ok   ")
            else:
                print(decalage+" Echec ")
    for k in range(len(a_choix)):
        v, i, j, a = a_choix[k]
        if k < len(a_choix)-1: # Il reste d'autres choix après
            print(f"{decalage}   \u251C\u2500{v}{fleche}{(i, j)}")
            affichage_arbre(a, decalage+"   \u2502 ", complet)
        else:  # C'est le dernier choix visité
            print(f"{decalage}   \u2514\u2500{v}{fleche}{(i, j)}")
            affichage_arbre(a, decalage+"     ", complet)
```

```
>>> a1 = [[(7, 5, 6), (9, 7, 1), echec], []]
>>> a2 = [[(8, 5, 6), (1, 7, 1), ok], []]
>>> arbre_test = [[(3, 0, 2), (5, 1, 7)], [(1, 2, 3, a1), (2, 2, 3, a2)]]
>>> affichage_arbre(arbre_test, complet=False)
   ├─1➔(2, 3)
   └─2➔(2, 3)
```

Le paramètre `complet` permet d'afficher toutes les valeurs faciles affectées ou juste les choix effectués.

```
>>> affichage_arbre(arbre_test)
3→(0, 2)
5→(1, 7)
    ├─1→(2, 3)
    │ 7→(5, 6)
    │ 9→(7, 1)
    │  Echec
    └─2→(2, 3)
      8→(5, 6)
      1→(7, 1)
        Ok
```

Voici la fonction récursive qui permet de faire la résolution, tout en construisant l'arbre, est donnée dans la page suivante.
La fonction qui permet de faire la résolution et l'affichage est :

```python
def resoudre2(grille, complet=True):
    arbre = []
    res = resoudre_rec2(grille, arbre)
    if not res or not verifier(grille):
        print("Il y a un problème")
    afficher(grille)
    affichage_arbre(arbre, "", complet)
```

Exercice 9 : Après avoir copié ces fonctions, regarder les arbres complets, ou non, obtenus pour des grilles faciles et des grilles diaboliques.

```python
def resoudre_rec2(grille, arbre):
    #print(grille)
    modifies = []
    coups_faciles = True
    faciles = []    # Liste des valeurs faciles placées
    while coups_faciles:
        coups_faciles = False
        restants = []
        for i in range(9):
            for j in range(9):
                if grille[i][j] == 0:
                    poss = possibles(grille, i, j)
                    if len(poss) == 1:
                        grille[i][j] = poss[0]
                        coups_faciles = True
                        modifies.append((i, j))
                        faciles.append((poss[0], i, j)) # On note la valeur
                    elif len(poss) > 1:
                        restants.append((len(poss), i, j))
                    else:
                        retirer(grille, modifies)
                        faciles.append(echec) # On note l'échec
                        arbre.append(faciles) # On met les valeurs faciles
                        arbre.append([]) # Et la liste des choix (vide)
                        return False
    if len(restants) == 0: # On a tout placé
        faciles.append(ok) # On note le succès
        arbre.append(faciles)
        arbre.append([]) # Les choix sont vides
        return True
    else:
        arbre.append(faciles)
        a_choix = []
        arbre.append(a_choix)
        restants.sort()
        _, i, j = restants[0]
        modifies.append((i, j))
        for v in possibles(grille, i, j):
            arbre2 = [] # L'arbre de résolution avec le choix fait
            grille[i][j] = v
            res = resoudre_rec2(grille, arbre2)
            a_choix.append((v, i, j, arbre2)) # On rajoute le choix à la liste
            if res:
                return True
        retirer(grille, modifies)
        return False
```