DS3info

16 mars 2024

La calculatrice est interdite. L'usage de tout document est interdit. La rigueur, le soin, la présentation seront fortement pris en compte dans la notation. Les résultats de chaque question seront encadrés.

Question de cours

On suppose que l'on dispose d'une fonction `Fusion(L:list,M:list)->list`, qui prend en paramètres deux listes triées et qui renvoie la fusion triée de ces deux listes. À l'aide de la fonction `Fusion`, écrire une fonction récursive `TriFusion(L:list)->list` qui renvoie la liste triée .

Sudoku

La résolution d'une grille de Sudoku est un casse-tête bien connu : à partir d'une grille presque vide, il est possible (pour une grille bien faite) de la compléter d'une manière unique. L'objectif de cet exercice est de mettre en œuvre deux méthodes permettant de compléter une grille de Sudoku, l'une naïve, et l'autre par backtracking. Une grille de Sudoku est une grille de taille 9×9 découpée en 9 carrés de taille 3×3 . Le but est de la remplir de chiffres entre 1 et 9, de sorte que chaque ligne, chaque colonne et chacun des carrés de taille 3×3 contienne une et une seule fois chaque entier de 1 à 9. On dira alors que la grille est bien remplie. En pratique, certaines cases sont déjà remplies et on fera l'hypothèse que le Sudoku qui nous intéresse est bien écrit, c'est-à-dire qu'il possède une unique solution.On représente en Python une grille de Sudoku par une liste de 9 listes de taille 9, dans laquelle les cases non remplies sont associées au chiffre 0. Ainsi, la grille suivante est représentée par la liste ci-contre :

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

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

Les 9 carrés de taille 3×3 sont numérotés du haut à gauche jusqu'en bas à droite. Ainsi, sur cette grille, le carré 0, en haut à gauche, contient les chiffres 6, 4 et 7 ; le carré 1, en haut au milieu, contient les chiffres 9, 2, 1 et 8 ; le carré 8 contient les chiffres 6, 2 et 1. On rappelle que les lignes du Sudoku sont alors les éléments de `L` accessible par `L[0]`, ... , `L[8]`. L'élément de la case (i, j) est accessible par `L[i][j]`.
On fera bien attention, dans l'ensemble du sujet, aux indices des listes ! Les lignes, ainsi que les colonnes et les carrés, sont indicées de 0 à 8.

Partie 1 : Généralités et fonctions annexes

1. Écrire une fonction `occurrences(elt,L)` qui renvoie le nombre d'occurrences de `elt` dans la liste `L`. Par exemple :

```
>>> occurrences(3, [1,7,3,4])
1
>>> occurrences(6, [1,7,3,4])
0
>>> occurrences(6, [1,6,6,4])
2
```

2. Déterminer la complexité de la fonction `occurrences` en fonction de `n = len(L)`. Justifier.
3. Donner un invariant de boucle qui permettrait de démontrer la correction de cette fonction (on ne demande pas la preuve de la correction).

On va commencer par écrire des fonctions permettant de vérifier si un Sudoku est bien rempli.

4. Écrire une fonction `ligneBienRemplie(L,i)` qui prend une liste de Sudoku `L` et un entier `i` entre 0 et 8, et renvoie `True` si la ligne est bien remplie et `False` sinon. On rappelle que la ligne `i` est bien remplie si elle contient exactement les chiffres de 1 à 9.
   On pourra utiliser, ou pas, la fonction `occurrences`.

5. Écrire de même une fonction `colonneBienRemplie(L,j)` pour la colonne `j`.

On définit de même (on ne demande pas l'écrire) la fonction `carréBienRempli(L,i)` pour le carré `i`.

6. Écrire une fonction `bienRempli(L)` qui prend une liste de Sudoku `L` comme argument, et qui renvoie `True` si la grille est bien remplie, `False` sinon.
7. Recopier et compléter la fonction suivante `ligne(L,i,j)` qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent sur la ligne d'indice `i` **en ne tenant pas compte de** `L[i][j]`.

```
def ligne(L,i,j) :
    chiffre = []
    for k in ..... :
        if ..... :
            chiffre.append(L[i][k])
    return chiffre
```

Ainsi, avec la grille donnée dans l'énoncé, on doit obtenir :

```
>>> ligne(L,0,0)
[6,2,5]
>>> ligne(L,0,1)
[2,5]
```

On définit, de la même manière, la fonction `colonne(L,i,j)` qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent dans la colonne `j` excepté `L[i][j]`, on ne demande pas d'écrire son code.

8. Que devrait renvoyer la commande `colonne(L,6,3)` ?

On se donne une case (i,j), avec (i,j) dans $\{0,\dots,8\}^2$. On admet que la case en haut à gauche du carré 3×3 auquel appartient la case (i,j) a pour coordonnées :

$$\left(3\times\left\lfloor\frac{i}{3}\right\rfloor, 3\times\left\lfloor\frac{j}{3}\right\rfloor\right)$$

Où $\lfloor x\rfloor$ représente la partie entière de x. Par exemple, le carré n°8 (en bas à droite) auquel appartient la case de coordonnées $(7,8)$ a pour case de coin en haut à gauche $\left(3\times\left\lfloor\frac{7}{3}\right\rfloor, 3\times\left\lfloor\frac{8}{3}\right\rfloor\right)=(6,6)$.

9. Recopier alors et compléter la fonction `carre(L,i,j)` qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent dans le carré 3×3 auquel appartient la case (i,j) toujours sans tenir compte de la case (i,j).

```
def carre(L,i,j) :
    icoin = 3*(i//3)
    jcoin = 3*(j//3)
    chiffre = []
    for x in range(.....) :
        for y in range(.....) :
            if ..... :
                chiffre.append(L[x][y])
    return chiffre
```

On rappelle que si `x` et `y` sont des entiers, `x//y` renvoie le quotient de la division euclidienne de `x` par `y`. Ainsi, avec la grille donnée dans l'énoncé, on doit obtenir :

```
>>> carre(L,4,6)
[9,7,3]
>>> carre(L,4,7)
[9,3]
```

10. Déduire des trois questions précédentes une fonction `conflit(L,i,j)` renvoyant la liste des chiffres que l'on ne peut pas écrire en case (i,j) sans contredire les règles du jeu. La liste envoyée peut très bien comporter des redondances. On ne prendra toujours pas en compte la valeur de `L[i][j]`.

11. Recopier alors et compléter la fonction `chiffresOk(L,i,j)` qui renvoie la liste des chiffres que l'on peut écrire en case (i, j).

```
def chiffresOk(L,i,j) :
    listeOk = []
    listeConf = conflit(L,i,j)
    for k in ..... :
        if ... :
            listeOk.append(k)
    return listeOk
```

Par exemple, avec la grille initiale :

```
>>> chiffresOk(L,4,2)
[2, 5, 8, 9]
```

On pourra/devra, dans la suite du sujet, utiliser librement les fonctions annexes définies précédemment comme si elles avaient été implémentées correctement.

PARTIE II - Algorithme naïf

Naïvement, on commence par compléter les cases n'ayant qu'une seule possibilité. Nous prendrons dans la suite comme exemple le Sudoku :

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

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

12. A partir des fonctions écrites dans la **Partie I**, écrire une fonction `nbPossible(L,i,j)` indiquant le nombre de chiffres possibles à la case (i, j).
13. On souhaite disposer de la fonction `unTour(L)` qui parcourt l'ensemble des cases du Sudoku et qui complète les cases dans le cas où il n'y a qu'un chiffre possible, et renvoie `True` s'il y a eu un changement, et `False` sinon. La liste `L` est alors modifiée par effet de bords. Par exemple, en partant de la grille initiale `M` :

```
>>> unTour(M)
True
>>> print(M)
M = [[2,0,0,0,9,0,3,0,0], [0,1,9,0,8,0,0,7,4],[0,0,8,4,0,0,6,2,9],
 [5,9,0,6,2,1,4,8,7], [0,2,7,0,3,8,1,6,5], [0,6,1,5,7,4,2,9,3],
[0,8,5,0,0,9,7,3,0], [9,3,6,0,5,0,8,4,2],[0,0,2,0,6,0,9,5,1] ]
```

On propose la fonction suivante :

```
def unTour(L):
    changement = False
    for i in range(1,9) :
        for j in range(1,9) :
            if L[i][j] = 0 :
                if nbPossible(L,i,j) = 1 :
                    L[i][j] = chiffresOk(L,i,j)[1]
    return changement
```

Recopier ce code en corrigeant les erreurs.

14. Écrire une fonction `complete(L)` qui exécute la fonction `unTour` tant qu'elle modifie la liste, et renvoie `True` si la grille est complétée, et `False` sinon.

Vous venez de coder un algorithme qui remplit automatiquement une grille de Sudoku !

Partie III - Backtracking

La deuxième idée est de résoudre la grille par `Backtracking` ou `retour-arrière`. L'objectif est d'essayer de compléter la grille de Sudoku en testant les combinaisons, en commençant par la première case, et jusqu'à la dernière. Si on obtient un conflit avec les règles, on est obligé de revenir en arrière. On va compléter la grille en utilisant l'ordre lexicographique, c'est à dire les cases $(0,0)$, $(0,1)$, $\dots (0,8)$, puis $(1,0)$, $(1,1),\dots(1,8)$, $(2,0),\dots(8,8)$. Considérons pour cette partie le Sudoku :

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

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

15. Écrire une fonction `caseSuivante(pos)` qui prend une liste `pos` qui est le couple des coordonnées de la case, et renvoie le couple des coordonnées de la case suivante en utilisant l'ordre lexicographique. Elle devra renvoyer `[9,0]` si `pos=[8,8]`. Par exemple :

```
>>> caseSuivante([1,3])
[1, 4]
>>> caseSuivante([1,8])
[2, 0]
>>> caseSuivante([8,8])
[9, 0]
```

La fonction principale va avoir la structure suivante :

```
def solution_sudoku(L):
    return backtracking (L,[0,0])
```

où `backtracking(L,pos)` est une fonction récursive qui doit renvoyer `True` s'il est possible de compléter la grille à partir des hypothèses faites sur les cases qui précèdent la case `pos`, et `False` dans le cas contraire. Ainsi :

- Si `pos` est la liste `[9,0]`, la grille est complétée, et on renvoie `True` (cas d'arrêt).
- Si la case est déjà remplie (donnée initiale du Sudoku), on passe à la case suivante via un appel récursif.
- Sinon, on affecte un des chiffres possibles à la case, et on passe à la case suivante par un appel récursif.

16. Compléter le squelette de la fonction `backtracking(L,pos)` selon les règles précédentes.

```
def backtracking(L, pos):
    """
    pos est une liste désignant une case du sudoku,
    [0,0] pour le coin en haut à gauche.
    """
    if (pos==[9,0]) :
        ....
    i, j=pos[0],pos[1]
```

```
if L[i][j] != 0 :
    return ....
for k in .... :
    L[i][j] = ....
    if .... :
        return ....
L[i][j] = ....
return ....
```

17. On suppose qu'au départ, il y a p cases déjà remplies. Montrer qu'au maximum, la fonction `backtracking` est appelée 9^{81-p} fois.