MPSI 2018-2019

Informatique

Concours blanc

La résolution d'une grille de Sudoku est une gymnastique du cerveau qui peut être assimilée à un décodage correcteur d'effacement. En effet, 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 taille 9×9, c'est-à-dire 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]`.

Remarque : on fera bien attention, dans l'ensemble du sujet, aux indices des listes. Les lignes, ainsi que les colonnes, sont indicées de 0 à 8.

Partie A. Généralités et fonctions annexes

Q1 Écrire une fonction `rechercheDansListe(elt, L)` qui renvoie la position de l'élément `elt` dans la liste `L`. Si l'élément n'est pas trouvé dans la liste, la fonction devra retourner -1. Par exemple

```
>>> rechercheDansListe(3, [1,7,3,4])
2
>>> rechercheDansListe(6, [1,7,3,4])
-1
```

Q2 Déterminer la complexité en temps de la fonction `rechercheDansListe` en fonction de $n = len(L)$ dans le pire des cas. Pour cela on comptera uniquement les comparaisons.

Q3 Si une grille de Sudoku est bien remplie, que peut-on dire de la somme de chaque ligne, chaque colonne et chaque carré de taille 3×3 ?

Q4 On va commencer par écrire des fonctions permettant de vérifier si un Sudoku est bien rempli.

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 définit de même (on ne demande pas les écrire) les fonctions `colonneBienRemplie(L,i)` pour la colonne i et `carréBienRempli(L,i)` pour le carré i.

Q5 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.

Q6 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 alors, 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).

Q7 On se donne une case (i, j), avec (i, j) dans $\{0, \ldots, 8\}^2$. On admet que la case en haut à gauche du carré 3×3 auquel appartient la case (i, j) a pour coordonnées :

$$\left(3 \times \left[\frac{i}{3}\right], 3 \times \left[\frac{j}{3}\right]\right)$$

où $[x]$ représente la partie entière de x.
Compléter alors 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]
```

Q8 Déduire des 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 pas en compte la valeur de `L[i][j]`

Q9 Compléter enfin 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 rechercheDansListe(..... :
            listeOk.append(k)
    return(listeOk)
```

Par exemple, avec la grille initiale :

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

On pourra, dans la suite du sujet, utiliser les fonctions annexes définies précédemment.

Partie B. Algorithme naïf

Naïvement, on commence par compléter les cases n'ayant qu'une seule possibilité. Nous prendrons dans la suite comme 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] ]
```

Q10 A partir des fonctions écrites dans la partie A, écrire une fonction `nbPossible(L,i,j)` indiquant le nombre de chiffres possibles à la case (i, j).

Q11 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
>>>
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. Vous mettrez les parties modifiées ou ajoutées d'une couleur différente.

Q12 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.

Partie C. 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] ]
```

Q13 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]
```

Q14 Écrire une fonction `caseLibreSuivante(pos, L)` qui à partir des coordonnées d'une case `pos` et du Sudoku `L` donne la prochaine case contenant 0 dans le sudoku.

```
>>> caseLibreSuivante([0,0])
[0, 3]
>>> caseLibreSuivante([1,6])
[2, 0]
>>> caseLibreSuivante([-1,8])
[0, 3]
>>> caseLibreSuivante([8,7])
[9, 0]
```

On définit alors, de la même manière, la fonction `caseLibrePrecedente(pos, L)` qui renvoie les coordonnées de la première case contenant 0 avant la position `pos`. S'il n'y en a pas, la fonction renvoie la position `[-1,8]`. On ne demande pas d'écrire son code.

Q15 On considère le programme suivant :

```
def solutionSudoku(L) :
    M = deepcopy(L)
    pos = caseLibreSuivante([-1,8], L)
    while pos[0]>=0 and pos[0]<=8 :
        Elt = M[pos[0]][pos[1]]
        M[pos[0]][pos[1]] = 0
        listeChiffresOk = chiffresOk(M, pos[0], pos[1])
        if len(listeChiffresOk)==0 :
            pos=caseLibrePrecedente(pos, L)
        elif Elt==0 :
            M[pos[0]][pos[1]]=listeChiffresOk[0]
            pos=caseLibreSuivante(pos, L)
        else :
            i = rechercheDansListe(Elt, listeChiffresOk)
            if i==len(listeChiffresOk)-1 :
                pos=caseLibrePrecedente(pos, L)
            else :
                M[pos[0]][pos[1]] = listeChiffresOk[i+1]
                pos=caseLibreSuivante(pos, L)
    return(M)
```

On rappelle que `M = deepcopy(L)` effectue une copie du Sudoku `L` et le met dans la variable `M`. Donner la valeur des variables à la fin de chacun des 5 premiers tours de la boucle while en remplissant le tableau suivant. Attention, pour la valeur de M, on ne demande que la première ligne.

Num. boucle	pos	Elt	listeChiffresOk	M[0]
0				
1				
2				
3				
4				
5				

où "Num. boucle" signifie le numéro du passage dans la boucle while. On rappelle que par convention, le numéro 0 est l'état des variables avant l'entrée dans la boucle.

Q16 Que peut-on dire du Sudoku si l'on sort de la boucle `while` avec `pos[0]<0` ? `pos[0]>8` ? Avec les hypothèses du problème, ne peut-on pas réduire ces conditions ?

Q17 Le but des dernières questions est de prouver l'algorithme `solutionSudoku`. Chaque tour de la boucle `while` sera repéré par le couple de variables `(M, pos)` pris en fin de boucle. Montrer que :
— Les cases strictement inférieures à `pos` ne contiennent pas de 0.
— Les cases strictement supérieures à `pos` et non pré-remplies sont toutes nulles.
— Les cases remplies vérifient les règles du Sudoku.

Q18 Pour chaque couple `(M, pos)`, on définit sa liste de référence `Lref(M, pos)` comme étant la liste des valeurs des cases du Sudoku de la position `[0,0]` à la position `pos` y compris, à laquelle on a ajouté en fin de liste l'entier 10. Ainsi par exemple, avec le Sudoku du début de la partie C, on a :

```
>>>Lref(M, [0,2])
[2,5,4,10]
>>>Lref(M, [0,3])
[2,5,4,0,10]
```

On dira que $(M_1, pos_1) \leq (M_2, pos_2)$ si et seulement si $Lref(M_1, pos_1) \leq Lref(M_2, pos_2)$; les listes de références étant comparées à l'aide de l'ordre lexicographique. Il est clair que la relation ainsi définie est réflexive et transitive. Montrer qu'elle est également antisymétrique sur l'ensemble des couples (M, pos) parcourus par l'algorithme.

Q19 Montrer qu'à chaque tour de boucle le couple (M, pos) est strictement plus grand qu'à la fin du tour précédent. En déduire que la boucle `while` se termine et que si en sortie, la valeur de `pos[0]` est `9`, le sudoku renvoyé est valide.

MPSI 2018-2019

Correction informatique

Concours blanc

Q1 La fonction `rechercheDansListe` peut s'écrire :

```
def rechercheDansListe(elt, L) :
    for i in range(0, len(L)) :
        if L[i]==elt :
            return(i)
    return(-1)
```

Q2 Dans le pire des cas, il y a n comparaisons. L'algorithme est en $O(n)$.

Q3 Si la grille est bien remplie, la somme des éléments d'une ligne est 1+2+3+4+5+6+7+8+9 puisque cette somme ne change pas si on trie les termes dans l'ordre croissant. De même pour les colonnes et les carrés 3×3

Q4 La fonction qui teste si une ligne est bien remplie peut s'écrire :

```
def ligneBienRemplie(L,i):
    chifTrouvé = [1]+[0]*9
    # chifTrouvé[i]=1 si le chiffre i est déjà apparu, 0 sinon.
    for j in range(0,9) :
        Chif = L[i][j]
        if chifTrouvé[Chif]==1 :
            return(False)
        else :
            chifTrouvé[Chif]=1
    return(True)
```

Q5 La fonction qui teste si une grille est bien remplie peut alors s'écrire :

```
def bienRempli(L):
    for i in range(0,9):
        if ligneBienRemplie(L,i)==False :
            return(False)
        if colonneBienRemplie(L,i)==False :
            return(False)
        if carreBienRempli(L,i)==False :
            return(False)
    return(True)
```

Q6 Le code de la fonction ligne de l'énoncé, une fois complété, ressemble à ceci :

```
def ligne(L,i,j):
    chiffre = []
    for k in range(0,9):
        if L[i][k]>0 and k!=j:
            chiffre.append(L[i][j])
    return chiffre
```

Q7 La fonction attendue dans l'énoncé est celle-ci :

```
def carre(L,i,j):
    icoin = 3*(i//3)
    jcoin = 3*(j//3)
    chiffre = []
    for x in range(icoin, icoin+3) :
        for y in range(jcoin, jcoin+3) :
            if (L[x][y] > 0 and (x,y)!=(i,j)):
                chiffre.append(L[x][y])
    return chiffre
```

Q8 La fonction conflit attendue dans l'énoncé peut ressembler à ceci :

```
def conflit(L,i,j):
    return(ligne(L,i,j)+colonne(L,i,j)+carre(L,i,j))
```

Q9 La fonction complétée ressemble maintenant à ceci :

```
def chiffresOk(L,i,j):
    listeOk = []
    listeConf = conflit(L,i,j)
    for k in range(1,10):
        if rechercheDansListe(k, listeConf)==-1 :
            listeOk.append(k)
    return(listeOk)
```

Q10 Les chiffres que l'on peut écrire à la case (i, j) peuvent alors être calculés avec

```
def nbPossible(L,i,j):
    return(len(chiffresOk(L,i,j)))
```

Q11 Les erreurs volontairement introduites concernaient l'indexation à partir de 0, le double-égal pour les tests et l'oubli de la modification de la variable `changement`. La fonction "corrigée" est celle-ci :

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

Q12 La fonction qui applique l'algorithme "naïf" est celle-ci :

```
def complete(L):
    pasFini = True
    while pasFini :
        pasFini = un_tour(L)
    return(bienRempli(L))
```

Q13 La fonction `caseSuivante` balaie, ligne par ligne, le sudoku :

```
def caseSuivante(pos):
    if pos[1]<8 :
        return([pos[0],pos[1]+1])
    else:
        return([pos[0]+1,0])
```

Q14 La fonction `caseLibreSuivante` peut s'écrire :

```
def caseLibreSuivante(pos, L) :
    pos2 = caseSuivante(pos)
    while pos2!=[9,0] and L[pos2[0]][pos2[1]]>0 :
        pos2 = caseSuivante(pos2)
    return(pos2)
```

Q15 Voici le tableau des valeurs des variables rempli.

Num. boucle	pos	Elt	listeChiffresOk	M[0]
0	[0, 3]	X	X	[2, 5, 4, 0, 9, 6, 3, 0, 0]
1	[0, 7]	0	[1, 7]	[2, 5, 4, 1, 9, 6, 3, 0, 0]
2	[0, 8]	0	[8]	[2, 5, 4, 1, 9, 6, 3, 8, 0]
3	[0, 7]	0	[]	[2, 5, 4, 1, 9, 6, 3, 0, 0]
4	[0, 3]	8	[8]	[2, 5, 4, 0, 9, 6, 3, 0, 0]
5	[0, 7]	1	[1, 7]	[2, 5, 4, 7, 9, 6, 3, 0, 0]

La variable `Elt` permet non seulement de mémoriser la case que l'on est en train de traiter, mais elle permet également de savoir si la position de la case précédente était plus grande ou plus petite que celle actuelle. En effet `Elt` est nul si on vient d'une case antérieure et non nul dans le cas contraire.

Q16 Si `pos[0]<0` en sortie de la boucle `while`, le sudoku n'a pas de solution. Si `pos[0]>8`, l'algorithme a trouvé une solution. Dans l'énoncé, on fait l'hypothèse que le Sudoku a une unique solution, on peut donc se contenter de `while pos[0]<=8`.

Q17 Montrons ces propriétés par récurrence.

Initialisation. L'instruction `pos = caseLibreSuivante([-1,8],L)` nous positionne sur la première case libre du Sudoku. Les cases qui précédent ne peuvent donc pas contenir de 0, les cases suivantes qui ne sont pas pré-remplies ne contiennent que des 0. Enfin, on a fait l'hypothèse que le Sudoku de départ vérifiait les règles.

Hérédité. On remarque que dans la boucle `while`, soit on met la case actuelle à 0 et on va vers la case précédente, soit on met un chiffre de la liste `chiffresOk` et on va vers la case suivante. Ces 2 actions conservent les 4 propriétés de l'énoncé.

Q18 Soient (M_1, pos_1) et (M_2, pos_2) ayant la même liste de référence. En particulier, ces listes sont de même longueur, on a donc $pos_1 = pos_2$. De plus, les Sudokus jusqu'à la position $pos_1 = pos_2$ sont identiques et après sont remplis de 0 d'après la question précédente. On a donc aussi $M_1 = M_2$.

Q19 Soit $L = [\dots, a, b, 10]$ la liste de référence à un instant donné. Il y a plusieurs possibilités :

- `len(listeChiffresOk)==0`. Dans ce cas L devient $L = [\dots, a, 10]$ qui est supérieur.
- `Elt==0`. Alors la liste L vaut $L = [\dots, a, 0, 10]$ et devient de la forme $L = [\dots, a, c, 0, 10]$, avec $c \in \{1, \dots, 9\}$ qui est supérieur.
- `Elt==0` et `i==len(listeChiffresOk)-1`. Comme dans le cas 1, la liste L devient $L = [\dots, a, 10]$ qui est toujours supérieur.
- `Elt==0` et `i!=len(listeChiffresOk)-1`. Enfin la liste devient $L = [\dots, a, c, 10]$ avec $c > b$ qui est donc supérieur.

Dans tous les cas la liste de référence croît strictement. Comme l'ensemble des listes de référence est fini, la boucle s'arrête. Si $pos[0]$ vaut 9 en sortie de boucle, on utilise la propriété 1 de la question 17 qui affirme que toutes les cases sont remplies et la propriété 4 pour la validité du Sudoku.