TP n=°1 : Sudoku

Informatique tronc commun 2ère année

N'oubliez pas de spécifier vos fonctions et de documenter votre code!

Pour trouver le dossier du TP, il faut aller dans :
- Mes dossiers
- -> Ce PC
- ->-> Documents Publics
- ->->-> par_professeur
- ->->->-> S puis SOULIER

1 Lecture de Python

```
1 def rec(li : list, x : int) -> bool :
2     for elem in li :
3         if elem == x :
4             return True
5     return False
```

1) Dans la fonction rec écrite à gauche, expliquer à quoi servent les : `list, : int` et `-> bool` dans la description des entrées.

2) Donner les valeurs que va prendre `elem`, la variable de boucle, dans l'appel de la fonction `rec([2,3,1,5,9,11],4)`

3) Expliquer ce que fait cette fonction.

2 Sudoku

Le sudoku est un jeu casse-tête qui consiste au remplissage d'une grille 9*9 à l'aide de chiffres. Certaines cases sont déjà préremplies et le but est de remplir les cases vides. Le remplissage ne doit pas être aléatoire et est régi par des règles, règles qui permettent de deviner le chiffre qui occupe une case vide. Ces règles sont :
- On remplit à l'aide des chiffres 1 à 9.
- Un chiffre doit être présent qu'une seule fois par ligne, colonne, et sous-carré 3*3.
- On ne peut pas modifier les chiffres déjà écrits.

On souhaite dans ce TP mettre en place une résolution informatique de ce problème. Pour cela, on modélise une grille de sudoku par une liste de listes d'entiers :
- Il y a 9 sous listes, représentant une ligne chacune.
- Les chiffres non pré-rempli sont représentés par un 0.
- Les chiffres remplis par un entier du nombre correspondant.

Pour la suite du sujet, toutes les variables nommées `sudoku` seront des listes de listes d'entiers représentant une matrice d'entiers de taille 9*9.

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

4) Donner les positions informatiques de tous les 9 présents dans la grille d'exemple.

5) Écrire une fonction `a_remplir(sudoku)` qui prend en entrée une grille de sudoku et renvoie une liste de toutes les positions qui sont encore à remplir.

6) Écrire une fonction `verif_lignes(sudoku,i)` qui vérifie si la ligne numéro `i` ne possède pas deux fois le même chiffre. On fera attention de ne pas automatiquement renvoyer `False` dès que la ligne contient plusieurs cases vides.
Faire de même une fonction `verif_colonne(sudoku,j)`.

7) Écrire une fonction `indice_carre(i,j)` qui renvoie la liste des éléments qui font partie du même sous carré que `(i,j)`.
En déduire une fonction `verif_carre(sudoku,i,j)` qui vérifie si le sous carré dont fait partie (i,j) ne contient pas de doublons.

8) Écrire une fonction `verif_sudo(sudoku)` qui vérifie si un sudoku respecte toutes les règles.

2.1 Méthode exhaustive

On va commencer par une première méthode dite exhaustive. Une méthode exhaustive consiste en la vérification de toutes les possibilités. Pour cela, notre méthode va créer une liste `l_choix` qui contiendra les choix faits pour remplir les cases vides :

Exemple : Si notre liste `a_remplir` contient `[(0,2),(0,4),(3,5)]` et que la liste `l_choix` contient `[8,3,4]`, cela indique que l'on est en train d'essayer de tester la solution où l'on a mis :

- un 8 en (0,2)
- un 3 en (0,4)
- un 4 en (3,5)

Si la solution marche, on la renvoie, si elle ne marche pas, on passe à la liste suivante `[9,3,4]`. Si cette nouvelle liste marche, on la renvoie, sinon, on passe à la liste `[1,4,4]`.

9) Écrire la fonction `incrementer(l_choix)` qui prend en entrée une liste `l_choix` et **modifie en place** la liste `l_choix` pour obtenir la liste suivante. Cette fonction **ne renvoie pas** de valeur.
On peut supposer que la liste `l_choix` n'est pas une liste `[9,9,9,9,9,9]`.

On est alors prêt à mettre en place notre recherche exhaustive de solutions. On va suivre la procédure suivante, à partir d'un sudoku donné en entrée :

- On commence par initialiser une liste `a_remp` qui contient toutes les cases à remplir de notre sudoku.
- On initialise une liste `l_choix` de même longueur qui ne contient que des 1.
- On remplit le sudoku pour que les cases vides contiennent les choix de la liste `l_choix`.
- Tant que notre sudoku n'est pas valide (début de boucle) :
  - ** On incrémente la liste `l_choix`.
  - ** On remplit le sudoku pour que les cases vides contiennent les choix de la liste `l_choix`.
- À la fin de la boucle, on renvoie les listes `l_choix` et `a_remplir`.

10) Écrire une fonction `exhaustif_sudoku(sudoku)` qui prend en entrée un sudoku et effectue la procédure décrite précédemment.

11) Tester sur le `sudoku1`. Quel est l'ordre de grandeur de la résolution exhaustive ?
Supprimer la ligne `sudoku1[8][8] = 2`, exécuter le fichier et retenter la résolution exhaustive du `sudoku1`. Quel est l'ordre de grandeur de la résolution ?

12) Donner le nombre de listes `l_choix` différentes en fonction de la taille de la liste `a_remp`. Est-il raisonnable d'utiliser cette méthode pour remplir un sudoku classique ?

2.2 Retour sur trace

Le retour sur trace (ou backtracking en anglais) est une méthode de recherche exhaustive optimisée. Elle s'applique lorsque le problème est un problème de satisfaction de contraintes, c'est-à-dire un problème qui est vérifié seulement si plusieurs contraintes sont vérifiées simultanément.

Le retour sur trace consiste à tester la contrainte liée à une case lorsque l'on teste un choix pour cette case. On pourra alors éliminer toutes les grilles qui contiennent ce sous-choix. Pour suivre les choix réalisés, on va utiliser un dictionnaire nommé `dic`, ainsi que deux listes de couples d'entiers, `l_choix` et `l_fait` , pour suivre les choix déjà faits et les choix encore à faire. On va tout d'abord écrire des fonctions intermédiaires.

13) Écrire une fonction `initialiser_choix(l_choix)` qui prend en entrée une liste de couples d'entiers, et qui renvoie un dictionnaire. Le dictionnaire contient comme clefs toutes les valeurs qui apparaissent dans `l_choix` et comme valeurs pour chacune de ces clefs l'entier 1.

L'une des étapes importantes du retour sur trace est la remontée des choix lorsque l'on trouve une contrainte non satisfaite. Le principe consiste à remonter jusqu'au dernier choix et à effectuer un test pour le choix suivant.

La remontée prend en entrée le `sudoku`, les listes `l_fait, l_choix` et le dictionnaire `dic`, et est décrite par la procédure suivante :

- On "pop" le dernier élément `(i,j)` de `l_fait`.
- On remet à 0 le chiffre en `(i,j)` dans le `sudoku`.
- On ajoute `(i,j)` dans la liste `l_choix`.

- On augmente de 1 la valeur associée à `(i,j)` dans `dic`.
- **Dans la cas où** cette valeur devient 10, on la met à 1 **et** on appelle récursivement la procédure avec les mêmes entrées.

14) Écrire la fonction `remonter(sudoku,l_fait,l_choix,dic)` qui suit la procédure décrite précédemment.

Maintenant qu'on a écrit une fonction pour effectuer la remontée du retour sur trace, on peut écrire la procédure du retour sur trace :

- Initialisation :
  - `l_choix` contient toutes les positions de cases à remplir.
  - `dic_choix`, le dictionnaire obtenu par la fonction de la question 15).
  - `l_fait`, une liste vide.
- Boucle tant que `l_choix` contient un élément :
  - On retire un élément `(i,j)` de `l_choix`, que l'on ajoute à `l_fait`.
  - On change la case `(i,j)` du sudoku par la valeur associée à `(i,j)` dans `dic`.
  - On vérifie si la ligne `i`, la colonne `j`, et le sous carrée `(i,j)` est toujours valide.
  - Si non, on effectue une "remontée".
- Retour : On renvoie le dictionnaire `dic_choix`.

15) Écrire une fonction `backtracking(sudoku)` qui met en place la procédure précédente.
Tester votre fonction sur le `sudoku1`, puis `sudoku2`.

Remarque : La complexité pire cas du backtracking est *la même* que la complexité pire cas de la recherche exhaustive. Seulement, en pratique, le backtracking est bien plus efficace.

2.3 Optimisation

Il est possible d'optimiser le backtracking. Pour cela, il faut que les premiers choix effectués par le retour sur trace soient les choix qui éliminent le plus rapidement possible les sous-remplissages.

16) Écrire une fonction `nombre_vide_voisine(sudoku)` qui prend en entrée un sudoku et renvoie un dictionnaire dont les clefs sont les cases à remplir. Les valeurs associées à une clef `(i,j)` est le nombre de cases vide dans la ligne `i`, la colonne `j` et le sous carré de `(i,j)`.

Une optimisation serait de commencer par les choix de ce dictionnaire associé aux valeurs les plus basses. Cette optimisation est utile seulement si les valeurs de ce dictionnaire sont très différentes.

2.4 Algorithme mains

L'algorithme à la main consiste en cette procédure :

- Tant qu'il reste une case à remplir :
  - On cherche une case où le remplissage est obligatoire
  - On remplit cette case avec la valeur correspondante.

17) Écrire une fonction `resoudre_mains(sudoku)` qui réalise la résolution d'un sudoku en suivant la procédure de résolution d'un humain.