Tours de Hanoï

Répondre aux trois premières questions posées dans l'énoncé proposé en cours (le cas de base est $n = 1$).

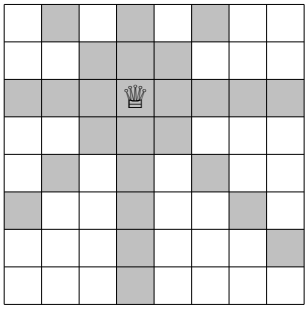
4. Calculer le nombre d'étapes nécessaires pour terminer le jeu avec n disques.

5. Initialement, la tour est : $[[2,1],[\,],[\,]]$. Donner les états des variables et les affichages lorsque l'on exécute `Hanoi(2,tour,0,1,2)`

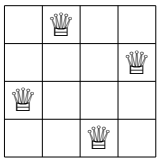
Backtracking

1. Le problème des 8 reines

On s'intéresse au problème suivant : sur un échiquier de n cases de côté, est-il possible de placer n reines sans qu'aucune d'elles ne puisse en prendre une autre ?
Une reine peut se déplacer sur l'échiquier soit horizontalement, soit verticalement, soit en diagonale : les cases grisées représentent les cases accessible par la reine sur un échiquier 8×8



Il s'agit donc de tenter de placer les n reines sans qu'il n'y en ait deux sur la même ligne, sur la même colonne ou sur une même diagonale. Voici une solution du problème pour $n = 4$:



On représentera l'échiquier par un tableau (type `numpy.array`) de taille $n \times n$ rempli de 0 et de 1 ; le chiffre 0 correspondant à une case vide alors que le chiffre 1 représente une case occupée par une reine. On placera donc d'abord les instructions :

```
import numpy as np
n = 4
A = np.zeros((n,n),int)
```

On part d'un échiquier vide et on essaie de placer les reines au fur et à mesure. Pour cela, on a besoin d'une fonction permettant de valider l'ajout d'une reine. Soit un échiquier valide (au maximum une reine sur chaque ligne, chaque colonne et chaque diagonale). On obtient un nouvel échiquier `A` en rajoutant une reine en position (i, j). Pour savoir s'il est valide, il suffit de calculer la somme des termes de la ligne i, de la colonne j, et de chaque diagonale passant par (i, j), puis de vérifier si chacune de ces sommes est $\leqslant 1$.

1. Écrire une fonction `sommeL(A,i)` prenant en argument un tableau `A` de taille $n \times n$ et un entier `i` $\in [\![0, n-1]\!]$ et qui renvoie la somme des éléments de la $i^{\text{ème}}$ ligne de `A`.

2. Écrire de même une fonction `sommeC(A,j)`, prenant les même arguments et qui calcule la somme des termes de la $j^{\text{ème}}$ colonne de `A`.

3. Écrire une fonction `sommeD(A,i,j)`, prenant en argument un tableau `A`, deux indices `i` et `j`, et qui renvoie les deux sommes des termes sur les lignes « diagonales »de `A` passant par la case de coordonnées (i, j) ; on pourra utiliser que la case (k, l) est sur une des lignes si et seulement si $i - k = \pm(j - l)$.

4. À l'aide des trois fonctions précédentes, écrire une fonction `valide(A,i,j)` qui prend en argument un tableau `A` et une position `(i,j)` et qui renvoie `False` s'il y a, sur l'échiquier représenté par `A`, plus d'une reine sur la ligne i, ou sur la colonne j, ou sur une des deux diagonales passant par (i, j), `True` sinon.

Le principe du backtracking (ou « retour sur trace ») est le suivant : en partant d'une situation donnée, on essaie de la compléter en une solution du problème par des essais successifs :

— Le cas de base est celui d'une solution complète du problème et alors on l'affiche.
— Sinon, on place successivement une reine sur chaque case de la première ligne vide de l'échiquier et pour chaque essai, on vérifie si l'échiquier reste conforme.
— Pour toutes les possibilités conformes, on recommence la démarche sur la ligne suivante (c'est une procédure récursive).
— Lorsque l'on a terminé d'examiner une situation, il faut penser à enlever la reine que l'on avait mise avant d'examiner la position suivante (c'est cette étape qui donne le nom « retour sur trace »).

5. On donne la fonction suivante :

```
1  def reines(A,i):
2      if i == n :
3          print(A)
4      else:
5          for j in range(n):
6              A[i,j] = 1
7              if valide(A,i,j):
8                  reines(A,i+1)
9              A[i,j] = 0
```

Exécuter `reines(A,0)` et commenter le résultat. Déterminer le rôle de chacune des lignes de cette fonction (si besoin, ne pas hésiter à rajouter des `print(...)` pour voir l'évolution de l'échiquier et comprendre le fonctionnement de cette fonction)

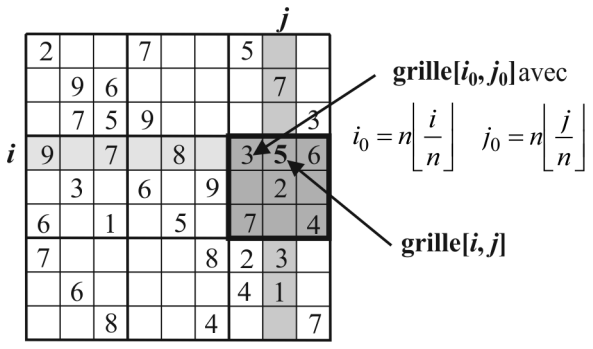
2. Sudoku

On cherche à compléter une grille de Sudoku : c'est une grille de taille 9×9 que l'on doit compléter avec les chiffres de 1 à 9 de sorte que :
— chaque ligne contienne une fois et une seule chaque chiffre de 1 à 9.
— chaque colonne contienne une fois et une seule chaque chiffre de 1 à 9.
— la grille est divisée en 9 blocs de taille 3×3 qui contiennent une fois et une seule les chiffres de 1 à 9.
On représentera la grille par un tableau numpy de taille 9×9, les cases vides étant remplies avec le chiffre 0. Par exemple :

```
import numpy as np
GrilleTest = np.array([[6,4,5,2,3,0,0,0,0],[0,0,7,0,0,0,0,5,4],
[8,1,9,4,5,0,2,3,0],[0,0,0,9,2,0,1,6,0],[0,0,0,0,6,0,0,0,0],
[0,0,6,7,8,1,0,2,9],[0,5,8,3,4,2,6,0,7],[2,7,0,0,0,0,8,4,0],[0,0,0,0,7,0,0,9,2]])
```

Pour pouvoir travailler avec des tableaux éventuellement plus petits, on envisagera par la suite des tableaux de taille $n^2 \times n^2$ (pour un Sudoku normal : $n = 3$, et pour un Sudoku "réduit" : $n = 2$). On cherche donc à remplir chaque ligne, chaque colonne et chaque bloc (de taille $n \times n$) avec les chiffres compris entre 1 et n^2.

Pour les trois premières fonctions demandées, on suppose que la grille de Sudoku est valide AVANT de placer un chiffre dans la ligne i et la colonne j. Le but de ces fonctions est donc de vérifier que la grille est toujours valide APRÈS avoir rempli la case (i, j).
On utilisera des boucles `while test` où le booléen `test`, initialement vrai, devient faux dès qu'une incompatibilité est détectée avec le chiffre placé en (i, j).



Pour cette partie, vous avez à votre disposition un fichier Python `backtracking_solution_partielle.py` *qui contient une partie de la solution (réponses aux questions de* ***1*** *à* ***5****).*

1. Écrire une fonction `valideL(grille,i,j)`, qui vérifie que la ligne `i` ne contient pas deux fois le chiffre placé dans la case `(i,j)`. La fonction renvoie `True` si la $i^{\text{ème}}$ ligne reste conforme, `False` sinon.

2. Écrire de même une fonction `valideC(grille,i,j)` qui effectue la même vérification sur la colonne `j`.

3. Écrire une fonction `valideB(grille,i,j)` permettant de tester si le bloc de taille $n \times n$ contenant la case `(i,j)` reste lui aussi conforme. On pourra balayer ce bloc en remarquant que ses cases ont pour coordonnées (i_0+k, j_0+m) avec $i_0 = n\left\lfloor \frac{i}{n} \right\rfloor$ et $j_0 = n\left\lfloor \frac{j}{n} \right\rfloor$, et $(k,m) \in [\![0, n-1]\!]^2$.

4. Écrire une fonction `valide(grille,i,j)` qui renvoie `True` si la ligne `i`, la colonne `j`, et le bloc contenant la case `(i,j)` sont toujours conformes aux règles du Sudoku après remplissage de la case case `(i,j)`, et `False` sinon.

5. Écrire une fonction `libre(grille)` qui prend en argument une `grille` et qui renvoie un couple d'indices (i,j) correspondant à une case vide de la grille s'il en existe. Dans le cas contraire, la fonction renvoie la chaîne de caractères `'rempli'`.

6. En adaptant le principe du backtracking exposé dans le problème des 8 reines, écrire une fonction `sudoku(grille)` qui prend en argument une `grille` à compléter (on supposera qu'elle peut effectivement être complétée) et qui renvoie les solutions du problème.

7. Tester votre fonction sur la grille `GrilleTest` donnée dans le fichier Python pour laquelle le temps de calcul est assez faible (quelques secondes), puis sur la grille `PbSudoku` donnée dans le fichier Python.

8. Comment pourrait-on améliorer la vitesse de résolution de cette méthode ?