# Résoudre les Sudoku

**Objectif :** L'objectif de ce tp est :

- d'utiliser le bon objet pour traiter un problème donné
- d'utiliser des algorithmes itératifs pour réduire le nombre de possibilités par case
- d'utiliser un algorithme récursif (back-tracking) pour traiter le cas où l'on est conduit à faire des hypothèses

## 1 Charger LA grille

Dans un premier temps, il convient de charger une grille.

**Q - 1** : *Créer une variable `Tab`, tableau de 9 lignes et 9 colonnes contenant la grille stockée dans le fichier `sudoku-0.txt`.*

**Q - 2** : *Choisir une structure de donnée `Grille` pour stocker un tableau 2D initialement composé de cellules vides.*

On considère alors que `Grille[i][j]` correspondra à la $i+1$ ligne et $j+1$ colonne de la grille à résoudre.

On veut associer pour chaque élément `Grille[i][j]` la liste des valeurs possibles. Certaines sont définies dès le départ par la grille vide. Les autres devront être éliminées au fur et à mesure par le programme.

Par exemple, la première valeur de la grille (ligne 1 et colonne1) n'est pas renseignée, on notera donc :

```
Grille[0][0]={1,2,3,4,5,6,7,8,9}
```

En revanche, ligne 1 et colonne 4, la grille contient la valeur 4. On aura donc : `Grille[0][3]={4}`

**Q - 3** : *Écrire une fonction `Initialisation(Mat,Tab)` qui associe à chaque élément `Mat[i][j]` soit un ensemble contenant les valeurs de 1 à 9, soit un ensemble contenant uniquement la valeur de la grille de départ.*

On dispose maintenant d'une variable `Grille` contenant l'ensemble des candidats sur chaque case de la grille. Il faut maintenant réduire la taille des ensembles de candidats.

## 2 Réduction du nombre potentiel de candidats

Si on travaille avec des listes ont a besoin des fonctions suivantes :

**Q - 4** : *Reconstruire une fonction `indice(T,x)` qui donne l'indice de `x` dans la liste `T` si `x` y est présent et renvoie -1 sinon. Le tableau est un tableau trié.*

Lorsqu'on a déterminé un chiffre dans la grille, il faut supprimer ce chiffre dans les ensembles de candidats sur la même ligne, la même colonne et le même groupe.

**Q - 5** : *Construire une fonction* `supprime(Mat,i,j)` *qui, si l'ensemble* `Mat[i][j]` *ne contient qu'une seule valeur (*`len(Mat[i][j])==1`*, supprime cette valeur (soit* `Mat[i][j][0]`*) des ensembles de la lignes i, de la colonne j et du bloc auquel appartient* `Mat[i][j]`*.*

**Remarque :** Faites attention à ne pas supprimer l'élément de `Mat[i][j]` en parcourant la ligne, la colonne ou le bloc !

A ce niveau, dès qu'on trouve une case dans la grille qui ne contient qu'une seule valeur, en appliquant la fonction `supprime(Mat,i,j)`, on allège la liste des candidats possibles sur les autres lignes.

**Q - 6** : *Construire une fonction* `simplification(Mat)` *qui applique la fonction* `supprime` *à chaque case de la grille n'ayant qu'un élément.*

**Q - 7** : *Exécuter les lignes suivantes :*

```
print(Mat)
simplification(Mat)
print(Mat)
simplification(Mat)
print(Mat)
simplification(Mat)
print(Mat)
simplification(Mat)
print(Mat)
```

C'est cool ! Il faut dire que ce sudoku n'était pas très difficile.

**Q - 8** : *Recommencer avec le fichier* `sudoku-1.txt`*.*

Il ne suffit plus de faire `simplification(Mat)`. Il va falloir tester les candidats restants.

## 3 Ordre de résolution de la grille

L'ordre de remplissage d'une grille de sudoku n'est pas forcément unique mais il est courant de commencer par les cases pour lesquels le choix est simple car il n'y a qu'un candidat. Les premières cases traitées sont celles de la grille de départ !

**Q - 9** : *Adapter la fonction* `Initialisation(Mat,Tab)` *pour construire une liste* `Ordre` *contenant 81 valeurs* `-1`*.*

On va remplir la liste `Ordre` afin qu'elle contienne l'ordre de remplissage des cases. L'élément `k` de la liste `Ordre` sera vu comme la cas $k+1$ de la grille en parcourant la grille ligne par ligne. Ainsi $k \in [\![0;80]\!]$ et $k = 9.i + j$.

Les cases non vides de la grille de départ prendront donc les premières valeurs, soit 0, 1, etc. Les cases vides sont initialement à -1. On crée alors l'indice `ind` qui indique le nombre de cases de la grille dont on connaît la valeur.

**Q - 10** : *Adapter la fonction* `simplification(Mat,Ordre)` *en* `simplification(Mat,Ordre,ind)` *pour qu'à chaque appel, pour tout élément* `M[i][j]` *ne contenant qu'un élément, si* `Ordre[i*9+j]==-1` *alors on appelle la fonction* `supprime(Mat,i,j)` *et* `Ordre[i*9+j]==ind` *puis* `ind` *est incrémenté de 1.*

**Q - 11** : *Afin de lancer l'outil de simplification jusqu'à stabilisation du nombre de candidats possible, écrire la fonction* `autosimplication(Mat,Ordre)` *qui appelle* `simplification(Mat,Ordre,ind)` *jusqu'à ce que ind n'évolue plus.*

A ce stade, `Mat` contient des listes réduites de candidats pour chaque case. L'algorithme s'étant terminé, il n'est plus possible de déterminer des solutions par suppressions de valeurs. Il convient de conserver la valeur de `ind` (on pose `ind1=ind`) pour les candidats potentiels restants. Le plus simple est de commencer par les cases qui ont le moins de candidats potentiels. On va donc finir de remplir la liste Ordre en recherchant toutes les cases à 2 candidats puis 3 candidats et ainsi de suite comme on l'a fait précédemment avec les cases ayant 1 candidat.

**Q - 12** : *Écrire la fonction* `Ariane(Mat,Ordre,ind1)` *qui permet de lister dans quel ordre les cases de la grille vont être parcourues.* `Ariane` *renvoie* `fil`*, c'est à dire la variable* `Ordre` *complétée.*

## 4 Backtracking

L'idée est la suivante :

- parcourir la liste Ordre à partir de `ind1`
- on teste les valeurs de 1 à 9
  - si la valeur est possible (ni présente sur la ligne, ni sur la colonne, ni sur le bloc) alors on l'inscrit dans la cellule et on passe à la cellule suivante.
  - sinon on remontre à la cellule suivante et on reprend le test des valeurs de 1 à 9 à partir de la valeur déjà inscrite dans la cellule

**Q - 13** : *Construire une variable* `Grille` *tel que* `Grille[i][j]=-1` *si* `len(Mat[i][j])>1` *et* `Grille[i][j]=Mat[i][j][0]` *sinon.*

Pour savoir si une valeur dans une case est possible, on crée 3 listes, `XL` pour ligne, `XC` pour colonne et `XB` pour bloc, qui indiquent si la valeur $k$ ($k \in [\![1;9]\!]$) est présente (sur la ligne, la colonne le bloc). Il s'agit donc de liste (indice ligne, colonne, bloc) de liste (valeurs de k de 1 à 9) de booleens (`True` et `False`) suivant la présente de $k$ sur la liste, la colonne ou le bloc.

**Q - 14** : *Adapter le programme de remplissage de la* `Grille` *pour mettre à jour* `XL`*,* `XC` *et* `XB`*.*

**Q - 15** : *Implémenter l'algorithme ci-contre.*

**Algorithm 1** Permet le remplissage recursif de la grille avec backtracking

```
    estvalide(Grille,position)
    entrée: un liste de liste 9,9 Grille et un entier position
    résultat: renvoie vrai, la grille est valide, faux sinon
 1: si position = 82 alors
 2:     renvoi: Vrai
 3: fin si
 4: n=fil[position]
 5: i,j=n//9,n-i*9
 6: pour k de 1 à 9 faire
 7:     si k n'est ni sur la ligne, ni sur la colonne, ni dans le bloc alors
 8:         ajouter k aux valeurs enregistrées XL, XC et XB
 9:         si estvalide(Grille,position+1) alors
10:             Grille[i,j]=k
11:             renvoi: Vrai
12:         fin si
13:         supprime k aux valeurs enregistrées XL, XC et XB
14:     fin si
15: fin pourrenvoi: Faux
```

**Q - 16** : *Sauvegarder la grille dans un fichier texte.*

**Q - 17** : *En introduisant des virgules entre chaque colonne, sauvegarder la grille dans un fichier* `.csv`*. L'ouvrir avec un tableur.*