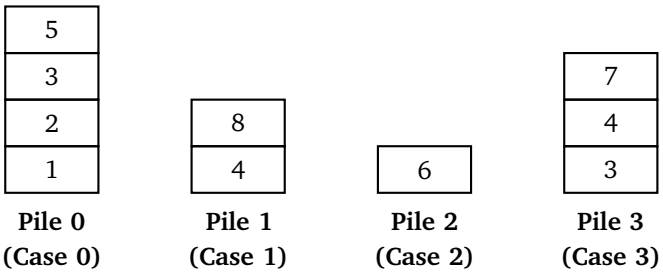
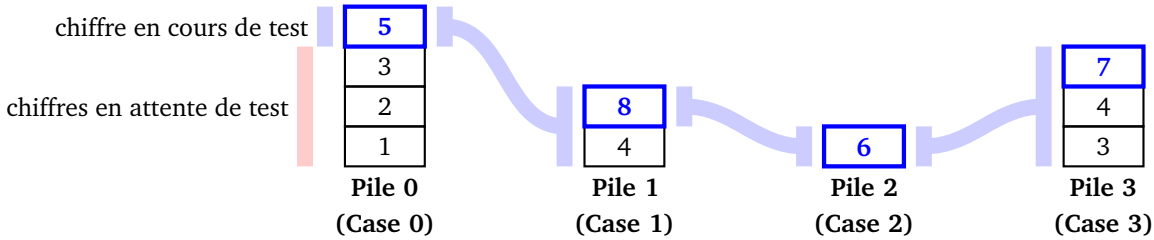
Sudoku

Tu vas programmer un algorithme qui complète entièrement une grille de sudoku. La méthode utilisée est la recherche par l'algorithme du « retour en arrière ».

Cours 1 (Recherche par retour en arrière)**.**
On cherche la solution à une grille de sudoku (les règles et les détails sont donnés dans l'activité 3). Chaque case correspond à une pile, chaque pile correspond à une liste de chiffres possibles.

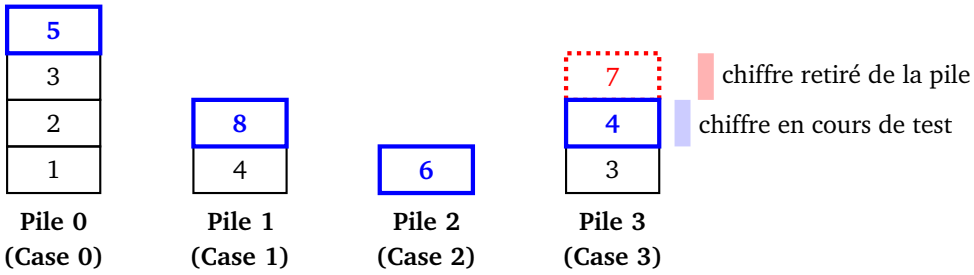


- On liste les chiffres possibles pour la case numéro 0 (par exemple la figure indique que les chiffres 1, 2, 3 ou 5 sont possibles).
- Parmi ces chiffres, on teste la configuration où le chiffre est le dernier de la liste (donc ici on suppose que dans la case numéro 0 on place le chiffre 5).
- Suivant cette hypothèse, on liste les choix possibles pour la case numéro 1 (ici 4 et 8 sont possibles). Attention, chaque pile dépend des piles précédentes. Ici pour la pile 0 formée de $[1, 2, 3, 5]$, avec 5 en haut, la pile 1 est $[4, 8]$.)
- On choisit pour cette pile numéro 1 de tester le dernier chiffre de la liste $[4, 8]$ (c'est comme si on plaçait 8 dans la case numéro 1).
- On continue. Pour les possibilités des cases suivantes, on tient compte de la configuration en cours de test (donc ici pour la case suivante, la figure ci-dessous avec les flèches indique que seul le chiffre 6 est possible, et cela tient compte du 5 en case 0 et du 8 en case 1).
- Si on continue et on arrive à remplir toute la grille, c'est gagné !



- Si à un moment on est bloqué, c'est-à-dire qu'il n'y a aucun chiffre possible pour la case suivante, alors on effectue un retour en arrière. Voyons un exemple. Partons de la configuration en cours de test

sur la figure : on a placé les chiffres 5, 8, 6 et 7 dans les quatre premières cases. Ce sont les chiffres en haut des piles.
Imaginons que pour la case suivante il n'y ait aucune possibilité. Alors on raye le chiffre 7 de la liste des possibilités pour la case numéro 3, c'est-à-dire qu'on le retire de la pile numéro 3. On suppose maintenant que le chiffre en cours de test pour la case numéro 3 est le chiffre 4. On repart en avant en cherchant les chiffres possibles pour la case suivante...



Cours 2 (Quatre problèmes simples)**.**
Les problèmes suivants n'ont pas d'intérêt en soi, mais ils sont beaucoup plus simples et vont nous permettre de tester notre algorithme de recherche par retour en arrière.

- **Problème 1.** Trouver toutes les listes de quatre chiffres telles que :
  — le chiffre au rang 0 est 1, 2, 3 ou 4 ;
  — le chiffre au rang 1 est 5 ou 6 ;
  — le chiffre au rang 2 est 7 ou 8 ;
  — le chiffre au rang 3 est 9.
  Exemple : $[3, 6, 7, 9]$ est une solution.
- **Problème 2.** Trouver toutes les listes de quatre chiffres telles que :
  — le chiffre au rang 0 est 1, 2 ou 3 ;
  — le chiffre au rang 1 est le double du chiffre au rang 0 ;
  — le chiffre au rang 2 est 5, 7 ou 9 ;
  — le chiffre au rang 3 est le même que le chiffre au rang 2.
  Exemple : $[2, 4, 7, 7]$ est une solution.
- **Problème 3.** Trouver toutes les listes de quatre chiffres telles que :
  — le chiffre au rang 0 est 1, 3, 5, 7 ou 9 ;
  — le chiffre au rang 1 est 2 ou 4 si le chiffre au rang 0 est supérieur ou égal à 5, et 6 ou 8 sinon ;
  — le chiffre au rang 2 est la moitié du chiffre au rang 1 ;
  — le chiffre au rang 3 est le chiffre au rang 0 diminué de 1 ou bien est 9.
  Exemple : $[5, 2, 1, 4]$ est une solution.
- **Problème 4.** Trouver toutes les listes de quatre chiffres telles que chacun des chiffres soit 0 ou 1 (sans autres contraintes). Exemple : $[0, 1, 0, 0]$ est une solution.

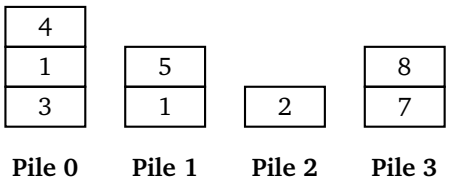
Activité 1 (Recherche par retour en arrière)**.**

Objectifs : programmer la recherche de solutions par la méthode du retour en arrière et l'appliquer ici à des problèmes simples.

On a besoin de deux variables globales :

- le nombre maximum de piles `n` (initialisé par exemple par `n = 4`),
- la liste des piles `les_piles`.

Par exemple la configuration des piles :



correspond à :

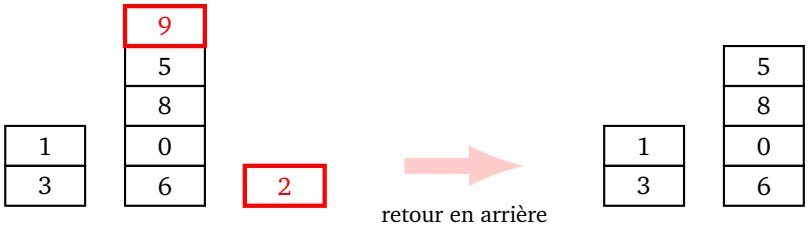
```
les_piles = [ [3,1,4], [1,5], [2], [7,8] ]
```

1. **Haut des piles.** Programme une fonction `haut_des_piles()` qui renvoie la liste des éléments en haut de chaque pile. Par exemple si `les_piles = [ [3,1,4], [1,5], [2], [7,8] ]` alors `haut_des_piles()` renvoie `[4, 5, 2, 8]`.
2. **Choix.** Programme une fonction `choix(i)` (qu'il faudra changer à chaque problème) et qui renvoie une liste afin d'initialiser la pile du rang i, c'est-à-dire la liste des choix possibles. Dans cette question, on répond au problème numéro 1, c'est-à-dire que `choix(i)` renvoie :
   - $[1,2,3,4]$ si $i = 0$,
   - $[5,6]$ si $i = 1$,
   - $[7,8]$ si $i = 2$,
   - $[9]$ si $i = 3$.

   Exemple. En partant d'une liste de piles vide `les_piles = []`, utilise cette fonction, pour arriver à la configuration où `les_piles` vaut `[ [1,2,3,4], [5,6], [7,8], [9] ]`.
3. **Retour en arrière.** Programme une fonction `retour()` qui exécute un retour en arrière dans notre liste des piles `les_piles`.

   On a besoin de faire un retour en arrière soit lorsque l'on se trouve dans une configuration bloquante, soit lorsque l'on a trouvé une solution et que l'on repart à la recherche d'une autre solution.
   - La situation la plus simple, c'est lorsqu'il reste au moins deux éléments sur la dernière pile : il suffit de retirer l'élément en haut de la dernière pile.

   - Par contre s'il n'y a qu'un seul élément sur la dernière pile, il faut supprimer cette pile et retirer un élément à la pile précédente.



   - Mais il se peut que sur plusieurs piles de la fin, il n'y ait qu'un seul élément. Il faut alors supprimer ces piles et retirer un élément sur une pile qui en contient deux.

Pour tenir compte de tous ces cas, l'algorithme est le suivant. Il fonctionne en deux étapes : tout d'abord on élimine (si besoin) toutes les piles de la fin qui sont seulement de taille 1 ; puis on retire l'élément du haut d'une pile de taille au moins 2.

Algorithme.
- Action : effectue un retour en arrière sur les piles, ce qui modifie la variable globale `les_piles`.
- On note r le rang de la dernière pile.
- Tant que $r \geqslant 0$ et que la taille de la pile numéro r vaut 1 :
  — supprimer la dernière pile,
  — faire $r \leftarrow r-1$
- Ensuite, si $r \geqslant 0$, supprimer l'élément du haut de la pile numéro r.

4. **Recherche des solutions.**

On recherche une configuration de n piles, qui va donner une solution en prenant chaque élément en haut des piles. Comment constituer la liste des piles ? La première pile est constituée des choix possibles pour le rang 0 à l'aide de la commande `choix(0)`. Tant que le nombre maximum de piles n'est pas atteint on essaie de rajouter une nouvelle pile (toujours avec la fonction `choix()`) si c'est possible on le fait, sinon on effectue un retour en arrière. Si on arrive au nombre maximal de pile, on a une solution. Si on arrive à une liste vide de piles (après des retours en arrière), c'est qu'on a testé toutes les possibilités.

Algorithme.
- — Action : recherche une, ou toutes les solutions, au problème déterminé par la fonction `choix()`. Une solution est formée de n éléments.
  — Sortie : affichage des solutions.
- On va utiliser la variable globale `les_piles`, initialisée par la liste vide.
- Ajouter dans `les_piles` une première pile donnée par `choix(0)`.
- On définit un drapeau `termine` à la valeur « Faux ».
- Tant que `termine` ne vaut pas « Vrai » :
  — Noter r la longueur de la liste des piles `les_piles`.
  — Si $r = 0$, alors mettre `termine` à la valeur « Vrai » (les piles sont toutes vides, il n'y a plus rien à tester).
  — Si $0 < r < n$ (le nombres maximum de piles n'est pas atteint) :
    — On calcule la prochaine pile `nouv_pile` par `choix(r)`.
    — Si `nouv_pile` n'est pas vide (c'est qu'il y a des possibilités), alors ajouter `nouv_pile` à la liste de toutes les piles `les_piles`,
    — sinon (il n'y a aucune possibilité, on est bloqué), effectuer un retour en arrière par la commande `retour()`.
  — Si $r = n$ (le nombre maximum de piles est atteint) alors on obtient une solution à notre problème en prenant chaque élément en haut des piles (par la fonction `haut_des_piles()`). On affiche cette solution. Puis on fait l'une des deux actions suivantes (commenter celle qui ne sert pas) :
    — si on cherche une seule solution, alors mettre `termine` à « Vrai » (on stoppe la recherche),
    — si on veut toutes les solutions, alors exécuter un retour en arrière par la commande `retour()`.

5. **D'autres problèmes.**

Modifie la fonction `choix()` afin de répondre aux problèmes donnés dans le cours auparavant.

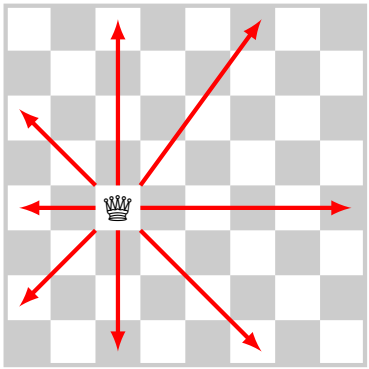
(a) **Problème numéro 2.**
- Il faut commencer par récupérer la liste des éléments en haut des piles renvoyée par la commande `haut = haut_des_piles()`.
- `choix(0)` donne la liste $[1,2,3]$,
- `choix(1)` donne la liste composée d'un seul élément : `[2*haut[0]]` (cela dépend donc de l'état des piles),
- `choix(2)` donne la liste $[5,7,9]$,
- `choix(3)` donne la liste composée d'un seul élément : `[haut[2]]` (cela dépend donc de l'état des piles).

(b) **Problème numéro 3.**
- Il faut commencer par récupérer la liste `haut` par la commande `haut_des_piles()`.
- `choix(0)` donne la liste $[1,3,5,7,9]$,
- `choix(1)` donne $[2,4]$ si `haut[0]` $\geqslant 5$, ou $[6,8]$ sinon,
- `choix(2)` donne la liste composée du seul élément `haut[1]//2`,
- `choix(3)` donne la liste composée des deux éléments `haut[0]-1` et 9.

(c) **Problème numéro 4.**

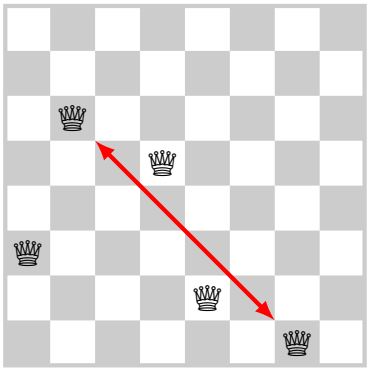
Comment fais-tu pour afficher tous les nombres binaires sous la forme de listes de 4 *bits* ? Tu dois afficher toutes les listes possibles : $[0,0,0,0]$, $[0,0,0,1]$,... jusqu'à $[1,1,1,1]$.

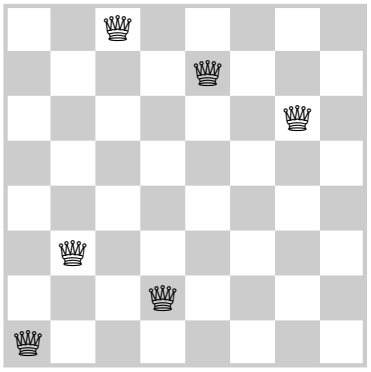
Cours 3 (Le problème des huit reines)**.**
Sur un échiquier de taille 8×8, une reine peut se positionner sur n'importe quelle case (noire ou blanche). Ensuite elle peut se déplacer pour capturer une pièce :
- sur n'importe quelle case de la même ligne,
- sur n'importe quelle case de la même colonne,
- sur n'importe quelle case d'une des deux diagonales.



Problème des huit reines. Placer 8 reines sur un échiquier de sorte qu'aucune reine ne puisse en capturer une autre.

Exemples. Voici à gauche un exemple d'une configuration qui ne conviendra pas. En effet, il y a deux reines situées sur une même diagonale. L'une peut donc capturer l'autre. À droite, un exemple de configuration avec 6 reines, aucune ne pouvant en capturer une autre. Comme il y a seulement 6 reines, et qu'il n'est pas possible d'en rajouter une autre, cela ne répond pas au problème.

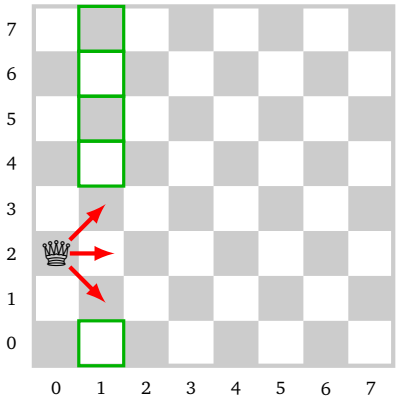


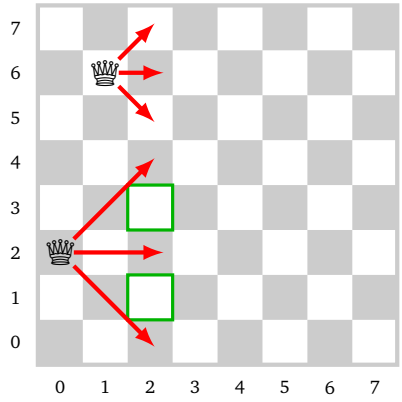


Méthode de recherche. Si on veut tester toutes les possibilités sans réfléchir, alors il y 64 possibilités pour placer la première reine, 63 pour la seconde... soit un nombre total de configurations qui vaut :

$$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178\,462\,987\,637\,760$$

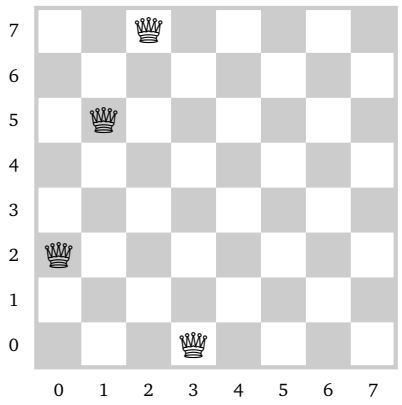
C'est beaucoup trop, même pour un ordinateur !
Pour diminuer le nombre de configurations à tester, nous allons tenir compte des reines en place, avant d'en ajouter une nouvelle. Pour cela on va placer une reine dans la colonne numéro 0, puis une reine dans la colonne numéro 1... (il ne peut y avoir qu'une seule reine par colonne).
L'idée est donc de tester une position pour la colonne numéro 0 (par exemple en ligne 2, voir les figures ci-dessous). On cherche ensuite quelles sont les positions possibles pour placer une reine sur la colonne numéro 1 : il y a 5 cases possibles (figure de gauche). Par exemple on choisit de placer une reine en colonne numéro 1, ligne 6 (figure de droite). Alors pour la colonne numéro 2, en tenant compte des deux reines déjà en place, il ne reste que 2 cases possibles. On retient que si on a déjà placé des reines, il reste un nombre assez limité de choix pour placer une reine dans la colonne suivante.





Modélisation.
On note n la taille de l'échiquier, qui est aussi le nombre de reines à placer. Les colonnes sont numérotées de $i = 0$ à $i = n-1$, les lignes aussi. On va placer une reine dans chaque colonne en partant de la gauche. Par exemple la configuration $(2, 5, 7, 0)$ signifie que l'on a placé :

- sur la colonne 0, une reine en ligne 2,
- sur la colonne 1, une reine en ligne 5,
- sur la colonne 2, une reine en ligne 7,
- sur la colonne 3, une reine en ligne 0.



Les configurations que l'on va tester sont toutes valides (aucune reine ne peut capturer une autre reine) mais pas forcément complètes (par exemple ici, il manque encore 4 reines).

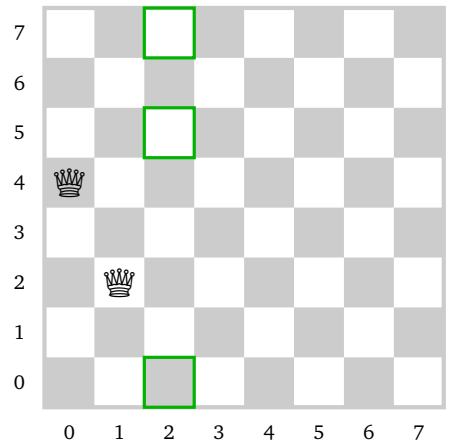
Activité 2 (Le problème des huit reines)**.**

Objectifs : écrire la fonction `choix()` *pour résoudre le problème des huit reines par la méthode du retour en arrière.*

- **Modélisation.**
  - Définis `n = 8`.
  - `les_piles` est la liste des piles. La pile numéro i contient une liste des lignes jouables pour la colonne numéro i.
  - Pour une configuration donnée, on initialise la pile suivante par la commande `choix(i)`, qui correspond à la colonne numéro i.
- **Fonction** `choix()`**.** Modifie la fonction `choix()` de l'activité précédente de sorte que `choix(i)` renvoie la liste des cases jouables sur la colonne numéro i en fonction de la configuration en cours de test (donnée par le haut des piles).
- **Indications.**
  - Récupère la configuration en cours de test par la commande `haut = haut_des_piles()`.
  - Quelles sont les lignes à éviter pour la prochaine colonne ?
    - Tu va inclure dans une liste `eviter` la liste des positions interdites pour cette colonne.
    - Il faut éviter les lignes déjà occupées par une reine (directement données par `haut`, tu peux donc initialiser `eviter` à `haut`).
    - Il faut éviter d'être sur une diagonale d'une autre reine : par exemple si on a une reine en colonne `j`, et donc en ligne `haut[j]`, alors, sur la nouvelle colonne `i`, les cases en ligne `haut[j]+i-j` et `haut[j]-i+j` sont interdites (si bien sûr, ces nombres sont compris entre 0 et $n-1$).
    - Les lignes possibles sont alors le complément des lignes à éviter. Ainsi `choix(i)` renvoie la liste des entiers de 0 à $n-1$ qui n'apparaissent pas dans la liste `eviter`.
  - Tu peux commencer par tester ton algorithme avec $n = 4$ (4 reines à placer sur un échiquier 4×4).
- **Exemples.**
  - `choix(0)` doit renvoyer `[0,1,2,...,7]` car il n'y a aucune contrainte.
  - Si par exemple :

    ```
    les_piles = [ [0,1,2,3,4], [1,2] ]
    ```

    Que doit renvoyer `choix(2)` ? La configuration en cours de test est donnée par le haut des piles, donc sur la colonne 0 on a placé une reine en ligne 4, et sur la colonne 1 une reine en ligne 2. La configuration testée est donc la suivante :



Alors la commande `choix(2)` doit renvoyer la liste $[0, 5, 7]$ qui sont les positions acceptables pour la colonne 2.

- **Questions.**
  — Combien y a-t-il de solutions différentes au problème des 8 reines ?
  — Combien y a-t-il de façons de placer 10 reines sur un échiquier 10×10 ?

Cours 4 (Sudoku)**.**
Grille de sudoku. Une grille de sudoku est une grille de taille 9×9 qu'il faut remplir des chiffres de 1 à 9 en respectant les règles suivantes :

- sur chaque ligne, chaque chiffre n'apparaît qu'une seule fois,
- sur chaque colonne, chaque chiffre n'apparaît qu'une seule fois,
- dans chaque sous-bloc 3×3, chaque chiffre n'apparaît qu'une seule fois.

Voici un exemple de grille à compléter (à gauche) et sa solution (à droite) :

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

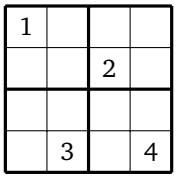
Grille de départ

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

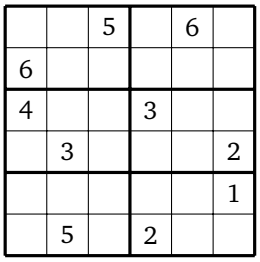
Grille résolue

Unicité. Une grille partielle de sudoku doit être conçue de sorte que la solution soit unique.

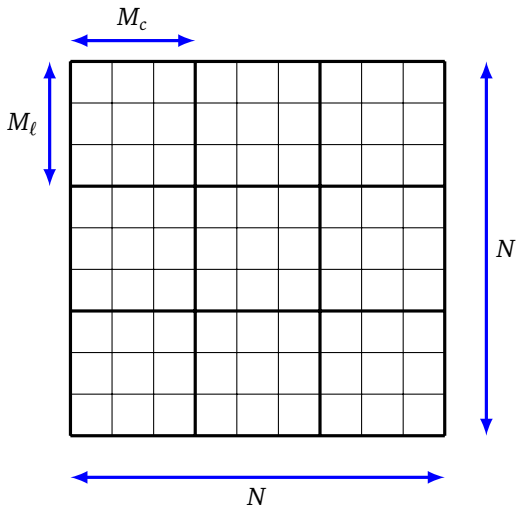
Modélisation. On va considérer des grilles $N \times N$, contenant donc $n = N^2$ cases, à remplir par des chiffres de 1 à N. La grille est découpée en sous-blocs de taille $M_\ell \times M_c$ (contenant chacun exactement N cases). On s'intéresse principalement aux grilles avec $N = 9$ et $M_\ell = M_c = 3$. Mais pour les tests on peut commencer par des grilles 4×4 ($N = 4$ et $M_\ell = M_c = 2$) ou des grilles 6×6 ($N = 6$ et $M_\ell = 2$, $M_c = 3$).



Grille 4 × 4

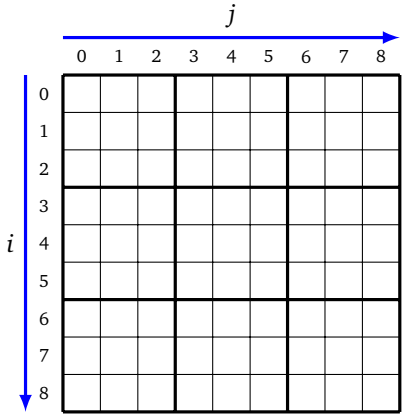


Grille 6 × 6



Numérotations des cases. Nous allons numéroter les cases de deux façons.

- *Coordonnées.* On peut identifier une case par ses coordonnées (i, j), i étant le numéro de ligne et j le numéro de colonne (les rangs commencent à 0 et finissent à $N-1$).
- *Ordre.* On numérote aussi les cases de $k = 0$ à $k = N^2 - 1$ en partant d'en haut à gauche.



Numérotation (i, j)

0	1	2	3	4	5	6	7	8
9	10	11	12					
18								
						61	62	63

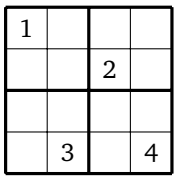
Numérotation k

- *Conversion.* Voici les formules qui permettent de passer de la numérotation par les coordonnées (i, j) à la numérotation par l'ordre k :

$$k = i \cdot N + j \qquad \text{et} \qquad \begin{cases} i &= k \,//\, N \\ j &= k \,\%\, N \end{cases}$$

Modélisation. On modélise une grille de sudoku par un tableau $N \times N$. On associe à une case vide le chiffre 0. Par exemple le tableau de gauche encode la grille de droite.

1	0	0	0
0	0	2	0
0	0	0	0
0	3	0	4



Piles. On va trouver la solution à une grille de sudoku par une recherche avec retour en arrière. Pour cela nous utiliserons une suite de piles, avec une pile par case (donc 64 pour une grille classique). Dans la pile numéro 0 on liste les chiffres possibles pour la case numéro 0, dans la pile numéro 1 on liste les chiffres possibles pour la case numéro 1, en tenant compte du chiffre en haut de la pile numéro 0, qui

correspond à la configuration en cours de test, dans la pile numéro 2 on liste les chiffres possibles pour la case numéro 2…
Pour une case remplie dans la grille de départ, la pile associée ne contient que le seul chiffre de cette case.

Exemple. Voyons le démarrage de la recherche d'une solution avec une grille 2 × 2.

1			
		2	
	3		4

Grille de départ

- La case numéro 0 contient un chiffre dans la grille de départ, donc la pile numéro 0 est réduite à la pile `[1]`. La liste des piles est pour l'instant réduite à une seule à pile : `[ [1] ]`.
- La commande `choix(1)` nous donne les possibilités pour la case numéro 1. Les seuls chiffres possibles sont 2 et 4, donc la pile numéro 1 est `[2,4]` et la liste des piles est maintenant `[ [1], [2,4] ]`.
- La configuration en cours de test est donnée par le haut des piles, donc pour l'instant c'est comme si la case numéro 1 contenait le chiffre 4 (grille de gauche ci-dessous).

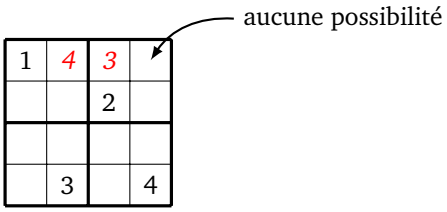
1	*4*		

Haut des piles

1	*4*		
		2	
	3		4

Grille en cours de test

- La commande `choix(2)` tient compte aussi des chiffres de départ, la configuration en cours de test est donc donnée par la grille de droite ci-dessus. La seule possibilité pour la case numéro 2 est le chiffre 3, la liste des piles est maintenant `[ [1], [2,4], [3] ]`.
- La configuration de la grille en cours de test est donc la grille ci-dessous. Pour la case suivante (la numéro 3) il n'y a aucune possibilité (`choix(3)` renvoie une liste vide).



Grille en cours de test

- Comme on est bloqué, pour continuer notre recherche on effectue un retour en arrière : ce qui nous ramène à la liste des piles valant : `[ [1], [2] ]`, c'est-à-dire à la configuration ci-dessous. Autrement dit, on a exclu la possibilité du chiffre 4 dans la case numéro 1.

1	*2*		
		2	
	3		4

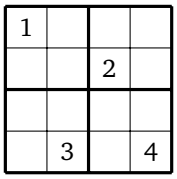
Grille après retour en arrière

- On liste alors les possibilités pour la case suivante (chiffre 3 ou 4 dans la case numéro 2), etc.

Activité 3 (Sudoku)**.**

Objectifs : programmer la résolution automatique d'une grille de sudoku.

1. **Grille de départ.**
   - Définis des variables globales `N`, `Ml`, `Mc`.
   - Initialise ce qui va être la grille de départ par :
     `grille_depart = [[0 for j in range(N)] for i in range(N)]`
   - Puis par des commandes du type `grille_depart[i][j] = valeur`, remplis la grille de départ.
   - Initialise toutes ces variables pour définir la grille de départ de l'exemple :
     `[[1, 0, 0, 0], [0, 0, 2, 0], [0, 0, 0, 0], [0, 3, 0, 4]]`



2. **Affichage.** Programme une fonction `voir_grille(grille)` qui affiche à l'écran une grille (éventuellement incomplète) de sudoku. Par exemple la grille de la question précédente peut s'afficher ainsi :

```
1 _ _ _
_ _ 2 _
_ _ _ _
_ 3 _ 4
```

3. **Conversions.** Programme deux fonctions `case_vers_numero(i,j)` et `numero_vers_case(k)` qui effectuent les conversions entre la numérotation par les coordonnées (i, j) et la numérotation par ordre k (voir les formules données dans le cours).

4. **Liste vers grille.** Programme une fonction `liste_vers_grille(liste)` qui à partir d'une liste de N^2 chiffres, renvoie une grille (sous la forme d'une liste de listes).

   Par exemple la liste :
   `[1, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 3, 0, 4]`
   devient la grille :
   `[[1, 0, 0, 0], [0, 0, 2, 0], [0, 0, 0, 0], [0, 3, 0, 4]]`

   Indication. Utilise la fonction `numero_vers_case()` après avoir initialisé une grille par
   `grille = [[0 for j in range(N)] for i in range(N)]`

5. **Chiffres en présence.** Programme des fonctions qui déterminent les chiffres déjà présents sur la grille :
   - `chiffres_ligne(i,grille)` renvoie la liste des chiffres présents sur la ligne i.
   - `chiffres_colonne(j,grille)` renvoie la liste des chiffres présents sur la colonne j.
   - `chiffres_bloc(i,j,grille)` renvoie la liste des chiffres présents dans le même sous-bloc que la case (i, j). (*Indication :* tu peux calculer les coordonnées (a, b) du coin supérieur gauche du sous-bloc par les formules : `a = Ml*(i//Ml)` et `b = Mc*(j//Mc)`.)

6. **Choix.** Programme la fonction `choix(k)` qui renvoie la liste des chiffres jouables pour la case numéro k, en tenant compte des chiffres de la grille de départ et de ceux la configuration en cours de test (donnée par les chiffres dans les cases numéro 0 à $k-1$).

   Méthode. Tout d'abord le nombre total de piles est $n = N^2$. Puis pour la fonction `choix(k)` :

- si la case numéro k contient déjà un chiffre de la grille de départ, on le conserve, la liste à renvoyer contient uniquement ce chiffre, c'est terminé.
- Sinon, il faut reconstituer une `grille`. Pour cela tu récupères d'abord la configuration en cours de test, par la commande `haut_des_piles()` qui contient la liste des chiffres des cases 0 à $k-1$ (à convertir en une grille $N \times N$). Ensuite il ne faut pas oublier d'ajouter les chiffres de `grille_depart`.
- Les chiffres à éviter pour la case k sont les chiffres présents sur la même ligne, ou la même colonne, ou dans le même sous-bloc. Les chiffres jouables sont les autres !

7. **Exemples.** Voici des exemples de sudokus de difficultés variées, sous la forme d'une liste de 64 nombres (à convertir en grille par la fonction `liste_vers_grille()`).

Facile

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

Moyen

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

Difficile

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

Variable

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

Cette dernière grille est déjà difficile (quelques secondes à résoudre), si on change la première ligne en :

```
8,1,0, 0,0,0, 0,0,0,
```

alors la résolution devient très difficile (plusieurs dizaines de secondes). Enfin, si on ne retient que le premier chiffre de la première ligne :

```
8,0,0, 0,0,0, 0,0,0,
```

notre programme échoue à trouver la solution en un temps raisonnable !