Chapitre 6

# Backtracking (retour sur trace)



## Introduction

Les **problèmes de satisfaction de contraintes** ou CSP (Constraint Satisfaction Problem) sont des problèmes mathématiques où nous cherchons des états ou des objets satisfaisant un certain nombre de contraintes ou de critères. Nous pouvons citer en exemple le remplissage d'une grille de sudoku, les contraintes sont simplement les règles du sudoku.

Le but de ce chapitre est de trouver une ou l'ensemble des solutions d'un problème de satisfaction de contraintes.

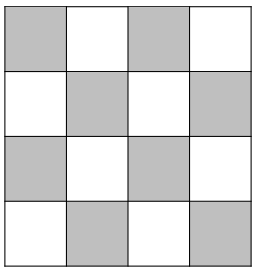
L'une des possibilités, appelée force brute, serait de remplir la grille entièrement et de tester a posteriori si la grille finale satisfait aux contraintes. Il faudrait alors tester toutes les possibilités, mais cela donnerait potentiellement $9^{81}$ tests à faire... beaucoup trop !

Une autre méthode consiste à travailler par essai-erreur, en testant des valeurs et en remplissant les cases de la grille dans l'ordre. Si les contraintes ne sont plus respectées, on s'arrête et on revient en arrière, d'où le nom de retour sur trace (backtracking).

Cette méthode utilise une représentation de l'espace des candidats sous la forme d'un arbre de recherche. Chaque solution partielle est un nœud de cet arbre. Chaque solution complète forme un chemin descendant de la racine à une feuille de l'arbre.

## 1 Premier exemple : le problème des quatre reines

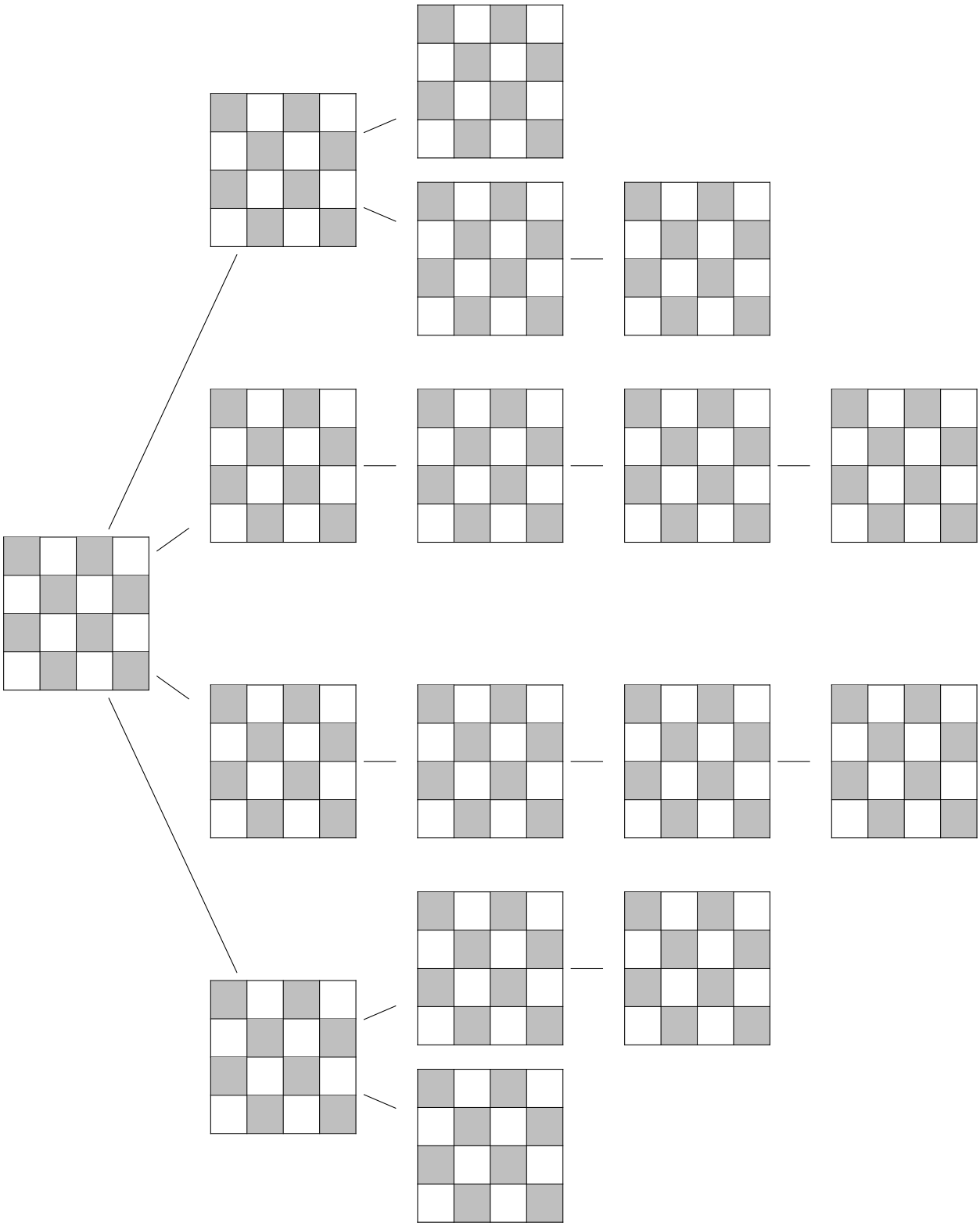
Nous avons à placer 4 reines sur un échiquier de $4 \times 4$ cases sans qu'elles se menacent entre elles.



De manière spontanée, nous sommes convaincus que chaque ligne contient une et une seule reine.

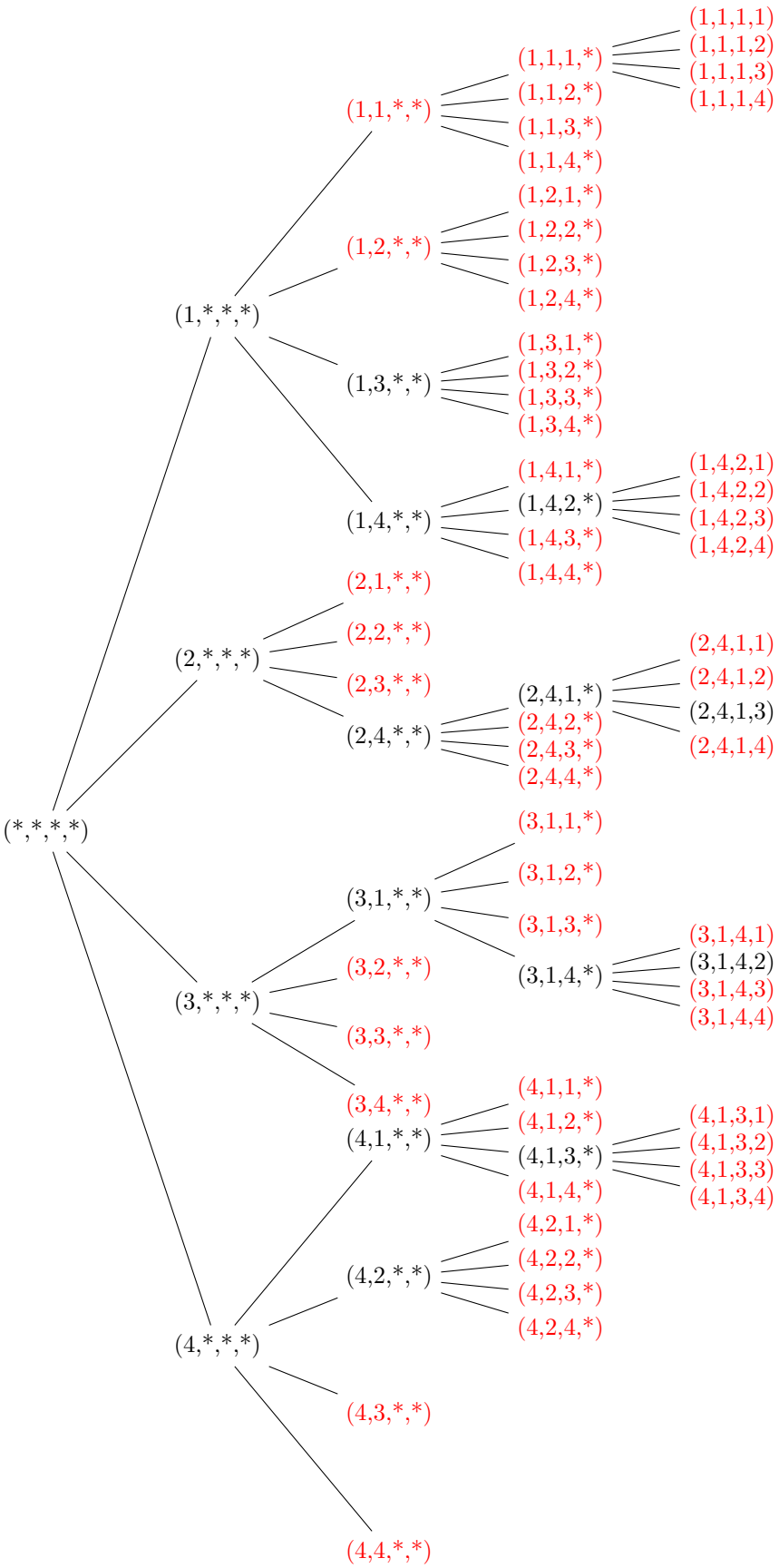
Nous pouvons donc donner les possibilités de placement de la reine sur la première ligne, puis, en fonction du placement précédent, donner les placements possibles de la reine sur la deuxième ligne; puis pour chaque configuration possible, placer la reine de la troisième, puis de la quatrième ligne.

Nous pouvons représenter les différentes possibilités de manière arborescente :



Dans cet arbre, les feuilles correspondant à des échiquiers non complets sont des solutions partielles impossibles à compléter. Le backtracking remonte alors dans l'arbre pour explorer d'autres solutions.

Si nous modélisons les placements possibles par un quadruplet tel que le $i$-ème élément est le numéro de la colonne où est placée la reine de la $i$-ème ligne, on obtient l'arbre suivant :



Les cas rouges sont les cas de blocage (non respect des contraintes) qui sont éliminés et pour lesquels on remonte (retour sur trace) pour explorer d'autres possibilités. Un tel arbre est appelé **arbre de recherche** du problème des 4 reines.

En pratique, les solutions partielles sont considérées comme des noeuds internes de l'arbre. Seules les solutions complètes non éliminées sont considérées comme des feuilles. Les feuilles sont donc les solutions du problème.

# 2 Cas général

## 2.1 L'algorithme de backtracking

Soit un problème $\mathscr{P}$, pour lequel la (les) solution(s) $\mathscr{S}$, si elle(s) existe(nt), se trouve(nt) dans un ensemble fini de candidats $\mathscr{E}$. On explore alors $\mathscr{E}$ (que l'on doit définir, ou qui est déjà défini) pour trouver la solution au problème $\mathscr{P}$.

L'algorithme de retour sur trace (backtracking) est une méthode systématique de parcours de l'espace $\mathscr{E}$ par essais successifs. On peut schématiser son fonctionnement de la manière suivante : à une étape donnée, l'algorithme est confronté à un certain nombre d'options et l'une d'entre elles est choisie. Ce choix entraîne, à l'étape suivante, un nouvel ensemble d'options qui dépend du choix qui a été effectué. Cette procédure est répétée jusqu'à atteindre un état final. Si la suite de choix est bonne jusqu'à obtenir une solution complète, on a trouvé une solution au problème posé. Dans le cas contraire, cet état ne convient pas et il faut remettre en cause (effacer) les choix précédemment effectués jusqu'à arriver à une situation de déblocage.

On peut représenter cette recherche par une exploration d'un arbre de recherche. L'algorithme débute à la racine et, à chaque nœud interne, on choisit l'un des fils pour explorer le sous-arbre correspondant jusqu'à arriver à une feuille. Si l'on est arrivé à une feuille en respectant toutes les contraintes, on a trouvé une solution du problème. Sinon, l'algorithme retourne en arrière pour continuer sa recherche en révoquant le choix le plus récent et en essayant un autre fils.

## 2.2 Implémentation de l'algorithme de backtracking

La pile est une structure de donnée fondamentale pour le backtracking, elle permet de stocker (empiler) toutes les solutions partielles possibles à tester et de les supprimer (dépiler) lors d'un retour.

Une implémentation naturelle de l'algorithme de backtracking consiste à utiliser la programmation récursive et sa pile d'exécution. En effet, si une solution partielle ne respecte pas les contraintes, on ne fait pas d'appel récursif et donc ce noeud interne est dépilé. Le retour sur trace se fait donc "tout seul".

# 3 D'autres exemples d'algorithmes de backtracking

## 3.1 Le problème SUBSET

Soit $X \subset \mathbb{N}$ et $S \in \mathbb{N}$. Le problème SUBSET consiste à chercher s'il existe un sous-ensemble d'éléments de $X$ de somme $S$. La liste des candidats est l'ensemble des sous-ensembles de $X$, et une approche par force brute devient vite impossible si le cardinal de $X$ est grand (car l'ensemble des candidats à tester est alors de cardinal $2^{|X|}$...).

On choisit alors une approche différente basée sur le principe du backtracking. Il existe deux cas triviaux :

- si $S = 0$, alors la réponse au problème SUBSET est Vrai (puisque $\emptyset$ est une solution) ;
- si $X = \emptyset$ et $S \neq 0$, alors la réponse au problème SUBSET est Faux.

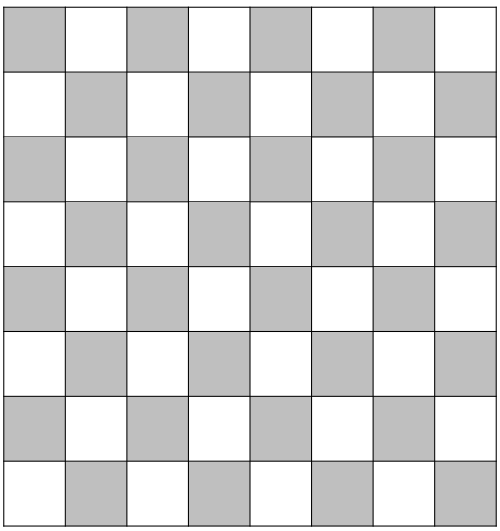
Pour le cas général, on adopte la stratégie suivante. Considérons $x \in X$. Il existe un sous-ensemble $Y$ de $X$ de somme $S$ si et seulement si :

- soit $x \in Y$ et les éléments de $Y$ sont de somme égale à $S$ : dans ce cas, il existe un sous-ensemble de $Y \setminus \{x\}$ de somme $S - x$ ;
- soit $x \notin Y$ et les éléments de $Y$ sont de somme égale à $S$.

Écrire une fonction en langage OCaml permettant de résoudre le problème SUBSET basé sur le principe du backtracking.

## 3.2 Le problème des huit reines

Nous cherchons toutes les configurations de 8 reines dans un échiquier de taille $8 \times 8$ telles que les reines ne se menacent pas entre elles.



On rappelle que la position des reines sur l'échiquier peut se représenter par un $N$-uplet où le premier élément du $N$-uplet nous indique la colonne de la reine dans la première ligne, le deuxième élément du $N$-uplet nous indique la colonne de la reine dans la deuxième ligne, ...etc...

Pour implémenter l'algorithme, nous allons mémoriser une position des 8 reines sur l'échiquier dans un tableau `position_reines` de longueur 8 tel que `position_reines.(i)` contient $-1$ si on n'a pas encore positionné la reine de la ligne numéro i, et contient le numéro de la colonne choisie si on a positionné la reine de la ligne numéro $i$.

Par exemple, l'échiquier représenté sur la figure ci-dessus est encodé par le tableau :

On considère la stratégie suivante : on place les reines sur l'échiquier ligne par ligne, en commençant par la ligne du haut. Pour placer la $i$-ème reine, on essaye les $n$ cases de la ligne $i$, de gauche à droite. Si une case particulière est menacée par une reine déjà placée, on l'ignore. Sinon, on place provisoirement la $i$-ème reine sur cette case et on cherche récursivement des placements cohérents des reines dans les lignes suivantes.

1. Écrire une fonction en langage OCaml `est_libre position_reines i j` qui, les lignes numéro 0 à $i-1$ ayant déjà été remplies, renvoie `true` si on peut placer une reine dans la colonne numéro $j$ et la ligne numéro $i$ et `false` sinon.

2. On considère la fonction en langage OCaml `affichage t` suivante, qui affiche à l'écran les éléments d'un tableau de longueur 8 sous forme d'un 8-uplet puis passe à la ligne :

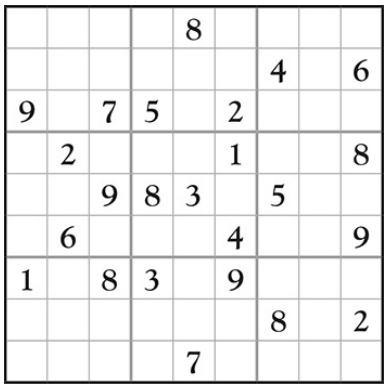
```
let affichage t =
  for k=0 to 6 do
    print_int t.(k);
    print_char ','
  done;
  print_int t.(7);
  print_newline()
;;
```

Écrire une fonction OCaml `placer_reines()` qui détermine toutes les positions possibles des 8 reines telles qu'elles ne se menacent pas entre elles, ainsi que le nombre de ces solutions.

On utilisera une fonction récursive auxiliaire `placer_reine_aux position_reines i c` qui, les reines des lignes numéro 0 à $i-1$ ayant déjà été placées, détermine et affiche tous les placements possibles des reines commençant par ces positions, et incrémente le compteur $c$ à chaque nouvelle solution trouvée.

## 3.3 La résolution d'une grille de Sudoku

Les règles du jeu sont supposées connues et nous est donc donnée notre grille de Sudoku à résoudre avec des cases préremplies.



Cette fois, nous cherchons s'il existe au moins une solution au sudoku et si oui, nous donnons une solution.

Nous sommes encore face à un problème de satisfaction avec contraintes, avec un arbre de recherche de taille impressionnante auquel nous allons appliquer le principe de backtracking. Nos choix successifs pouvant nous conduire à une impasse, c'est-à-dire une grille ne respectant pas les règles du sudoku, nous remonterons alors dans l'arbre de recherche jusqu'à pouvoir tester une nouvelle possibilité.

Le parcours dans l'arbre de recherche se matérialise par la lecture de la grille de sudoku case par case. Nous choisissons arbitrairement de lire la grille ligne par ligne, une ligne étant lue de gauche à droite.

1. **Structures de données.**

   Une grille sera naturellement représentée par un tableau de 9 lignes et 9 colonnes, les indices variant alors de 0 à 8.

   Une case d'une grille de sudoku peut être vide, contenir une valeur imposée, se voir affecté une valeur définitive ou non. Aussi, nous allons définir le type construit :

   ```
   type element = Vide | Fixe of int | Variable of int ;;
   ```

   Une grille de sudoku nous amène à définir le type :

   ```
   type grille = element array array ;;
   ```

   Par exemple pour la grille ci-dessus :

   ```
   let grille_ex = Array.make_matrix 9 9 Vide ;;
   grille_ex.(0).(4) <- Fixe 8 ;
   grille_ex.(1).(6) <- Fixe 4 ;
   grille_ex.(1).(8) <- Fixe 6 ;
   grille_ex.(2).(0) <- Fixe 9 ;
   grille_ex.(2).(2) <- Fixe 7 ;
   grille_ex.(2).(3) <- Fixe 5 ;
   grille_ex.(2).(5) <- Fixe 2 ;
   grille_ex.(3).(1) <- Fixe 2 ;
   grille_ex.(3).(5) <- Fixe 1 ;
   grille_ex.(3).(8) <- Fixe 8 ;
   grille_ex.(4).(2) <- Fixe 9 ;
   grille_ex.(4).(3) <- Fixe 8 ;
   grille_ex.(4).(4) <- Fixe 3 ;
   grille_ex.(4).(6) <- Fixe 5 ;
   grille_ex.(5).(1) <- Fixe 6 ;
   grille_ex.(5).(5) <- Fixe 4 ;
   grille_ex.(5).(8) <- Fixe 9 ;
   grille_ex.(6).(0) <- Fixe 1 ;
   grille_ex.(6).(2) <- Fixe 8 ;
   grille_ex.(6).(3) <- Fixe 3 ;
   grille_ex.(6).(5) <- Fixe 9 ;
   grille_ex.(7).(6) <- Fixe 8 ;
   grille_ex.(7).(8) <- Fixe 2 ;
   grille_ex.(8).(4) <- Fixe 7 ;;
   ```

2. **Quelques fonctions utiles pour la suite.**

   (a) Écrire une fonction en langage OCaml

   ```
   int_of_elt :  element -> int
   ```

   qui prend en argument une variable de type `element` et renvoie la valeur entière correspondante. Le cas d'une case vide renvoie un message d'erreur.

   (b) Écrire une fonction en langage OCaml

   ```
   case_suivante :  int*int -> int*int
   ```

   qui prend en argument une case (qui n'est pas la dernière case de la grille) définie par sa position `(a,b)`, numéro de ligne et numéro de colonne, et renvoie la position de la case suivante dans la lecture de la grille.

3. **Recherche des valeurs à tester dans une case donnée.**

   (a) Écrire une fonction en langage OCaml

   ```
   valeurs_interdites_ligne :  grille -> int -> int -> int list
   ```

   qui prend en argument une grille de sudoku et une position `(i,j)` dans la grille, et renvoie la liste des valeurs interdites en position `(i,j)` par règle de la ligne.

(b) Écrire une fonction en langage OCaml

```
valeurs_interdites_colonne :  grille -> int -> int -> int list
```

qui prend en argument une grille de sudoku et une position `(i,j)` dans la grille, et renvoie la liste des valeurs interdites en position `(i,j)` par règle de la colonne.

(c) Écrire une fonction en langage OCaml

```
valeurs_interdites_bloc :  grille -> int -> int -> int list
```

qui prend en argument une grille de sudoku et une position `(i,j)` dans la grille, et renvoie la liste des valeurs interdites en position `(i,j)` par règle du bloc.

(d) Écrire une fonction en langage OCaml

```
valeurs_permises :  grille -> int -> int -> int list
```

qui prend en argument une grille de sudoku et une position `(i,j)` dans la grille, et renvoie la liste des valeurs à tester en position `(i,j)`.

4. **Résolution du Sudoku.**

(a) Écrire une fonction récursive en langage OCaml

```
sudoku_aux :  grille -> int -> int -> bool
```

qui renvoie `True` si l'on a réussi à compléter toute la grille à partir des hypothèses faites dans les cases précédant $(i, j)$ (une valeur a donc déjà été affectée à chaque case précédant $(i, j)$) et `False` dans le cas contraire.

Le cas de base est lorsqu'on a réussi à remplir la grille (c'est-à-dire lorsque $i = 9$ et $j = 0$). Dans ce cas, on renvoie `True`. Sinon,

- si la case $(i, j)$ est déja remplie, il n'y a rien à faire, on passe à la case suivante ;
- sinon, on essaie successivement toutes les valeurs autorisées en case $(i, j)$ : pour chacune de ces valeurs, on écrit un chiffre en case $(i, j)$ et on fait un appel récursif à `sudoku_aux`.

  On doit donc définir une autre fonction récursive auxiliaire

  ```
  parcours_valeurs_permises :  grille -> int list -> int -> int -> bool
  ```

  qui prend aussi en entrée la liste de toutes les valeurs à tester en case $(i, j)$. Cette fonction renvoie `True` si on a réussi à remplir la grille au moins une fois, et donc `False` si aucune possibilité ne convient.

(b) En déduire une fonction en langage OCaml

```
sudoku :  grille -> bool*grille
```

qui renvoie un booléen indiquant si on a réussi à remplir au moins une fois la grille et renvoie également l'état final de la grille.