Première partie : problème des n reines

Le problème des n reines consiste à placer n reines sur un damier de taille $n \times n$ de sorte qu'aucune reine ne menace directement, au sens des échecs, l'une des autres reines (autrement dit, il ne peut y avoir qu'une seule reine sur chaque ligne, sur chaque colonne et sur chaque diagonale).

Dans un carré de taille n, les n lignes doivent donc avoir une unique reine dont la position de colonne peut être repérée par un indice $1 \leq j \leq n$. On représentera une solution du problème des n reines par un tableau de taille n dont la valeur de la i-ème case, $1 \leq i \leq n$, est celle de l'indice de la reine de ligne i.

Question 1
Ecrire une fonction de retour sur trace `nreines : int -> int array` prenant en argument un entier n renvoyant une solution du problème des n reines. En cas d'absence de solution, on renverra un message d'erreur.

Question 2
Estimer la complexité de votre algorithme. Essayer le sur différentes valeurs.

Deuxième partie : problème du Sudoku

Une grille de Sudoku est un tableau de taille 9×9 subdivisé en 9 carrés 3×3. Le but est de compléter une grille partiellement préremplie de chiffres de 1 à 9 de sorte que chaque ligne, chaque colonne et chaque carré comporte une unique occurrence de chacun des neuf chiffres. Une "bonne" grille doit posséder une unique solution.

Une telle grille sera représentée par un tableau 9×9 d'entiers. Une case non remplie contiendra la valeur 0.

Question 3
Ecrire une fonction `possible_ligne : int array array -> int -> int -> bool` prenant en argument une grille de Sudoku, un entier $0 \leq i \leq 8$ et un chiffre $1 \leq k \leq 9$ indiquant si le chiffre k est déjà présent dans la grille sur la ligne i.

Question 4
Ecrire une fonction `possible_colonne : int array array -> int -> int -> bool` prenant en argument une grille de Sudoku, un entier $0 \leq j \leq 8$ et un chiffre $1 \leq k \leq 9$ indiquant si le chiffre k est déjà présent dans la grille sur la colonne j.

Question 5
Ecrire une fonction `possible_bloc : int array array -> int -> int -> int -> bool` prenant en argument une grille de Sudoku, deux entiers $0 \leq i, j \leq 8$ et un chiffre $1 \leq k \leq 9$ indiquant si le chiffre k est déjà présent sur le bloc de la grille auquel appartient le couple (i, j).

Question 6
Ecrire une fonction `possible : int array array -> int -> int -> int -> bool` prenant en argument une grille de Sudoku, deux entiers $0 \leq i, j \leq 8$ et un chiffre $1 \leq k \leq 9$, indiquant si la case (i, j) peut recevoir le chiffre k (sans créer de conflit en ligne, colonne ou bloc).

Question 7
Ecrire une fonction de retour sur trace `completer : int array array -> int array array` prenant en argument une ligne de Sudoku préremplie la complétant en respectant les règles.

Question 8
Essayer cette fonction sur plusieurs grilles, par exemple les grilles

```
g1=[| [| 5; 0; 0; 0; 0; 0; 0; 0; 4|] ; [| 3; 2; 0; 6; 0; 0; 0; 0; 9|] ; [| 0; 0; 0; 2; 0; 0; 8; 3;
0|] ; [| 2; 0; 0; 0; 8; 0; 0; 4; 0|] ; [| 0; 4; 0; 0; 2; 0; 9; 0; 5|] ; [| 0; 0; 8; 0; 0; 7; 0; 2;
1|] ; [| 0; 0; 0; 0; 5; 0; 7; 1; 0|] ; [| 0; 0; 0; 0; 6; 0; 0; 9; 0|] ; [| 0; 0; 0; 8; 0; 9; 0; 0;
2|] |]
g2=[| 1; 2; 3; 0; 0; 0; 0; 8; 0|] ; [| 0; 0; 0; 0; 0; 0; 2; 0; 9|] ; [| 0; 0; 0; 0; 6; 0; 7; 0; 0|]
; [| 0; 7; 0; 0; 0; 3; 0; 6; 0|] ; [| 9; 0; 1; 0; 0; 0; 3; 0; 0|] ; [| 0; 3; 8; 0; 5; 4; 0; 0; 0|]
; [| 3; 0; 0; 7; 0; 0; 0; 0; 1|] ; [| 0; 0; 0; 0; 4; 5; 0; 0; 6|] ; [| 0; 6; 5; 0; 0; 0; 0; 7; 3|]
|]
g3=[| [|0; 0; 9; 0; 0; 2; 0; 0; 0|] ; [| 1; 6; 0; 0; 5; 0; 0; 0; 0|] ; [| 0; 0; 0; 0; 7; 0; 0; 0;
2|] ; [| 0; 0; 0; 0; 0; 0; 0; 0; 5|] ; [| 0; 5; 0; 8; 0; 3; 0; 9; 0|] ; [| 6; 0; 7; 0; 9; 0; 2; 4;
0|] ; [| 0; 0; 0; 0; 0; 1; 6; 0; 0|] ; [| 8; 9; 0; 0; 0; 0; 3; 0; 0|] ; [| 0; 0; 0; 4; 0; 0; 0; 0;
0|] |]
```

Troisième partie : Problème du cavalier

Ce problème consiste à faire parcourir à un cavalier un damier de taille $n \times n$ en ne le faisant passer qu'une unique fois sur chaque case. Le cavalier doit se déplacer en suivant les règles de déplacement du jeu d'échecs.

Question 9

Ecrire une fonction de retour sur trace prenant en argument un entier n renvoyant, si possible, la liste des positions (i, j), $1 \leq i, j \leq n$, suivies par un cavalier parcourant une et une seule fois chaque case d'un damier $n \times n$.

Question 10

Evaluer la complexité de votre algorithme. Comparer la avec celle du problème des n reines. On pourra notamment essayer sur un damier de taille 8×8.

Pour améliorer quelque peu la complexité de l'approche précédente, nous allons plutôt commencer par essayer de déplacer le cavalier sur les cases où le nombre de déplacements sera faible. Pour ce faire, on pourra utiliser un tableau de taille $n \times n$ dont la case d'indice (i, j) contient le nombre de déplacements possibles pour le cavalier.

Question 11

Ecrire une fonction `deplacements_cavalier : int -> int array array` prenant en argument un entier n renvoyant le tableau à deux indices dont l'élément d'indice (i, i) contient le nombre de déplacements possibles du cavalier depuis la case (i, j).

Outre le nombre de déplacements, on utilisera la liste des déplacements possibles.

Question 12

Ecrire une fonction `liste_deplacements : int -> int*int list array array` prenant en argument un entier n renvoyant le tableau dont l'élément d'indice (i, j) contiendra la liste des déplacements possibles ordonnée par nombre de déplacements possibles.

Lorsque l'on tentera un nouveau déplacement du cavalier, on choisira d'abord pour nouvelle position celle qui a le moins de déplacements possibles. On modifiera en conséquence les tableau des nombres de déplacements et des déplacements possibles (le nombre de déplacements d'une case parcourue sera 0).

Question 13

Ecrire une nouvelle fonction résolvant le problème du cavalier par retour sur trace en appliquant la stratégie précédente. L'essayer et la comparer à la fonction précédente.