

Première 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 1

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 2

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 3

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 4

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 5

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 6

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|] |]
```

Deuxième 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 et en le faisant revenir à la case initiale après qu'il soit passé par les autres cases. Le cavalier doit se déplacer en suivant les règles de déplacement du jeu d'échecs.

On représente les cases du damier par des couples $(i, j) \in \llbracket 0, n-1 \rrbracket$. Un déplacement sera un couple (x, y) , x étant la variations d'abscisse, y la variation d'ordonnée. $x > 0$ (resp. $y > 0$) consiste en un déplacement vers la droite (resp. le bas), $x < 0$ (resp. $y < 0$) consiste en un déplacement vers la gauche (resp. le haut).

Question 7

Ecrire une constante `déplacements` qui est la liste des déplacements possibles d'un cavalier.

Pour mémoriser les déplacements du cavalier, on utilisera un tableau $n * n$ de couples (x, y) . La case d'indice (i, j) contient la case (x, y) vers laquelle s'est déplacé le cavalier depuis (i, j) . Lorsque le cavalier ne s'est pas encore rendu sur la case (i, j) , celle-ci contiendra le couple $(-1, -1)$. En outre, pour savoir lorsqu'un échiquier est complet (donc que toutes les cases ont été parcourues), on utilisera une variable entière mémorisant le nombre de cases non remplies. On utilisera donc le type

```
type echiquier = { damier : (int*int) array array ; mutable nbre_cases_vides : int};;
```

Question 8

Ecrire une fonction `possible : echiquier -> (int*int) -> (int*int) -> bool` prenant en argument un échiquier, un couple représentant la position actuelle du cavalier, un couple représentant un déplacement, un entier représentant la taille du damier renvoyant le booléen indiquant si le déplacement du cavalier est une case de l'échiquier et si cette case est non parcourue.

Question 9

Ecrire une fonction `cases_atteignables : echiquier -> (int*int) -> (int*int) list` renvoyant la liste des cases que peut atteindre le cavalier depuis une case. Elle aura comme arguments un échiquier et une position.

Lors du fonctionnement de la fonction de retour sur trace et de son exploration de l'arbre des solutions potentielles, on aura besoin de renvoyer une solution si elle existe, une information exprimant la non existence dans une branche de l'arbre de solution sinon. Pour que ces deux réponses soient du même type, on définit le type suivant :

```
type solution =  
  | None  
  | Solution of (int*int) array array;;
```

Question 10

Ecrire une fonction `retour_sur_trace : echiquier -> (int*int) -> (int*int) list -> solution` qui prend en argument un échiquier, une position (celle du cavalier), une liste de déplacements possibles, qui essaye pour chaque déplacement de compléter l'échiquier. On pensera à chaque itération à vérifier si l'échiquier est complet et à supprimer le déplacement effectué en cas d'échec de la tentative.

Question 11

Ecrire une fonction `cavalier : int -> solution` qui prend en argument un entier et qui renvoie une solution du problème du cavalier s'il en existe une.