Dans ce TP nous allons nous intéresser aux algorithmes de « retour sur trace » ou « backtracking ». Le principe de ces algorithmes est de parcourir intégralement un arbre de décision en profondeur à la recherche d'une solution d'un problème. En cas d'impossibilité, on fait « demi-tour » d'où le nom de ces algorithmes.

I - Résolution d'un sudoku

Nous allons voir comment il est possible de résoudre un sudoku par backtracking. Commençons par rappeler rapidement le principe du jeu.

On considère une grille carré ayant 9 cases de cotés. Le but est de remplir chaque case de la grille avec un nombre allant de 1 à 9 de telle sorte que :

- (L) : Un même nombre apparait une fois et une seule dans chaque ligne.
- (C) : Un même nombre apparait une fois et une seule dans chaque colonne.
- (B) : Un même nombre apparait une fois et une seule dans chacun des 9 blocs.

On dispose généralement d'une grille partiellement remplie et on veut compléter les cases qui sont vides en respectant les règles (L), (C) et (B).

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

On va vouloir repérer les cases de la grille de deux manières :

- Par un numéro allant de 0 à 80. La case 0 est la case en haut à gauche, la case 1 celle qui est juste à droite et ainsi de suite, la case 80 est la case en bas à droite.
- Par un couple (i, j) où i est le numéro de la ligne et j celui de la colonne. Les deux vont des 0 à 8.

1) Écrire une fonction `numero : int*int -> int` tel que `numero (i,j)` renvoie le numéro de la case à la ligne `i` et la colonne `j`

2) Écrire une fonction `coord : int-> int*int` tel que `coord x` renvoie le couple `(i,j)` qui sont les coordonnées de la case numérotée x.

3) Les contraintes (L), (C) et (B) peuvent se visualiser par un graphe. Précisément, on considère le graphe $G = (S, A)$ non orienté où
   - l'ensemble des sommets S est $[\![0, 80]\!]$; chaque sommet correspond à une case
   - pour tout couple $(x, y) \in S^2$, le couple (x, y) appartient à A si et seulement si $x \neq y$ et si l'une des contraintes (L), (C) ou (B) impose que le nombre de la case x soit différent du nombre de la case y.
     Par exemple $(2, 5) \in A$ car les cases de numéro 2 et de numéro 5 sont sur la même ligne ; de même $(37, 45) \in A$ car les cases de numéro 37 et de numéro 47 sont dans le même bloc.

   Écrire une fonction `construitGraphe : unit -> int array array` telle que `construitGraphe ()` renvoie le graphe G ci-dessus. On utilisera une implémentation par une matrice d'adjacence.

On suppose maintenant que le graphe est construit. Il ne sera pas modifié et sera stocké pour la suite de ce TP dans une variable globale `g`.

4) On veut maintenant remplir la grille de sudoku. Pour cela on va utiliser les types suivants

```
type nombre = Vide | Nb of int;;
type grille = nombre array array
```

Une grille de sudoku sera une matrice de `nombre` avec 9 lignes et 9 colonnes. Si un case contient `Vide` cela signifie que la case n'a pas encore de nombre affecté.

Le but des questions suivante est de compléter une grille initiale en une grille complète.

a) Écrire une fonction `verification : grille -> int -> int -> bool` telle que pour toute grille, `verification grille x n` renvoie un booléen selon que l'on peut affecter à la case dont le numéro est `x` le nombre `Nb(n)` sans enfreindre les contraintes (L), (C), (B).

b) Écrire une fonction `caseVide : grille -> int` telle que `caseVide grille` renvoie le numéro d'une case vide de la grille. On pourra, par exemple renvoyer la case vide de plus petit numéro. Si la grille est entièrement remplie, la fonction renverra 81.

c) Le principe de la fonction de backtracking est le suivant :
- La fonction trouve une case qui est encore vide ; s'il n'y en a plus, elle lève une exception `Fini`
- La fonction essaye de mettre toutes les valeurs possible dans cette case (en utilisant la fonction `verification`) ; une fois qu'elle a mis la nouvelle valeur, elle se rappelle pour aller chercher un nouvelle case vide
- Quand on arrive à une impossibilité, elle revient en arrière en remettant la case concernée à `Vide`

Écrire une fonction `sudoku : grille -> grille` qui permet de résoudre un sudoku par backtracking.

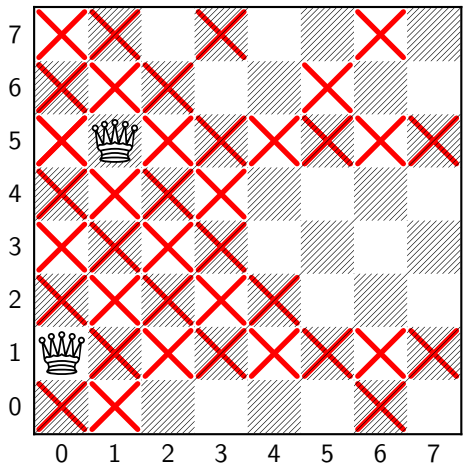
d) Tester la fonction en commençant par une grille vide. Tester ensuite avec la grille proposée en introduction ; on pourra se simplifier la vie en écrivant d'abord une fonction `textToGrille : string -> grille`.

II - Problème des huit reines

On dispose d'un échiquier de 8×8 cases. On veut placer dessus 8 reines de telle sorte qu'aucune n'en menace une autre.

On rappelle qu'aux échecs, une reines peut se déplacer dans sa ligne, sa colonne ou en diagonale.

Le diagramme ci-dessous montre les cases interdites après avoir posé deux reines sur l'échiquier.



On va coder les solution par un tableau `s` comprenant 8 cases tel que pour tout `j` compris entre 0 et 7, `s.(j)` soit le numéro de la ligne où est placée la reine de la i-ème colonne.

Quand une solution n'est pas complète, on indiquera `-1` dans la case correspondant à une colonne où il n'y a pas encore de reine. Par exemple la situation de l'exemple ci-dessus est

```
s = [| 1 ; 5 ; -1 ; -1 ; -1 ; -1 ; -1 ; -1 |]
```

5) Écrire une fonction `reine : unit -> int array` tel que l'appel `reine ()` renvoie une solution au problème.

   On pourra s'inspirer de ce qui a été mis en place pour les sudokus et écrire une fonction

   `verif : int array -> int -> int -> bool`

   telle que `verif s j i` renvoie un booléen selon que, avec la position des reines décrites par le tableau `s`, on puisse poser une dame à la colonne `j` et la ligne `i`.

6) Le mathématicien Gauss a calculé en 1850 qu'il y avait 72 configurations possibles. Avait-il raison ?

7) Généraliser les fonctions précédentes pour calculer le nombre de possibilités de placer n reines sur un échiquier de $n \times n$ cases. Ne pas tester avec de trop grandes valeurs de n.