TP n°2 - Retour sur trace

1. Problème des n reines

On rappelle le problème des n reines qui, pour un entier $n > 0$ donné, revient à placer n reines sur un échiquier de $n \times n$ cases sans que les reines ne puissent s'attaquer mutuellement, conformément aux règles de déplacement de cette pièce aux échecs.

On modélise le problème avec n variables $x_0, \ldots, x_{n-1}$ représentant pour chaque ligne la colonne à laquelle la reine est (on sait qu'il faut une reine par ligne pour résoudre le problème). Les domaines sont donc tous $D = [|0, n-1|]$.

Les contraintes sont les suivantes :

Pour tous $i, j \in \{0, \ldots, n-1\}$,

$$i \neq j \Rightarrow x_i \neq x_j, \text{ (pas deux reines sur la même colonne)}$$
$$i \neq j \Rightarrow x_i + j \neq x_j + i, \text{ (pas deux reines sur la même diagonale descendante)}$$
$$i \neq j \Rightarrow x_i - j \neq x_j - i. \text{ (pas deux reines sur la même diagonale ascendante)}$$

On représentera d'abord un candidat total par un tableau `tab` de type `int array` et de taille n.

1. Écrire une fonction `conflit : int -> int -> int -> int -> bool` qui teste si deux reines peuvent s'attaquer à partir de leur numéros de ligne et de colonne.
2. Écrire une fonction `est_solution : int array -> bool` qui teste si un candidat total est une solution (totale).

Notre but est d'abord de programmer une recherche exhaustive de solutions. Pour cela on vous fournit une fonction `succ` qui a un candidat (total) associe le suivant dans l'ordre lexicographique.

3. Programmer une fonction exhaustive `nb_solutions : int -> int` qui compte le nombre de solutions au problème des n reines.
   On cherchera le plus grand entier n pour lequel le programme répond rapidement (quelques secondes).

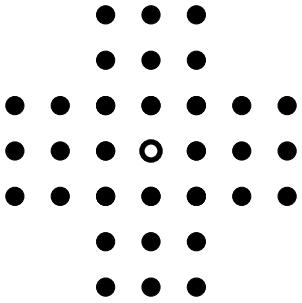
Maintenant on cherche à écrire une fonction fonctionnant par retour sur trace.

On représentera maintenant les candidats partiels et totals par une tableau `tab` de type `int option array` et de taille n.

4. Écrire une fonction `est_valide : int -> int option array -> bool` qui dit si le candidat partiel où les i premières reines ont été placées dans le tableau est une solution partielle.
5. En utilisant le principe du retour sur trace et la fonction `est_valide`, écrire une fonction `nb_solutions2 : int -> int` qui compte le nombre de solutions au problème des n reines.
   On cherchera le plus grand entier n pour lequel le programme répond en quelques secondes, pour comparer à la méthode exhaustive.

2. Jeu du solitaire

On considère ici le jeu du solitaire. On a un plateau comportant 33 emplacements et initialement 32 pions, représenté par des ronds noirs, et un emplacement vide au centre (blanc) :



Les différents mouvements possibles consiste à passer d'une configuration ●●○ à ○○● et ainsi à diminuer d'un pion le nombre total de pions.

Ces configurations peuvent être rencontrées dans les directions horizontales ou verticales.

On considère que la partie est gagnée quand il n'y a plus qu'un pion sur le plateau.

Première implémentation

On va représenter un plateau par le type OCaml :

```
type case = Vide | Pion | Invalide
type plateau = case array array
let n = 7
(* Un plateau est une matrice 7x7 avec des cases invalides aux coins *)
```

6. Définir une fonction **print_plateau** qui affiche un plateau sous un format textuel lisible.
7. Écrire une fonction **plateau_initial : unit -> plateau** qui renvoie un plateau correspondant à la configuration de départ.

Un mouvement peut être assimilé à un triplet de coordonnées décrivant, dans l'ordre ••• les trois cases concernées. Il se trouve que la case centrale est toujours le milieu des deux autres, on peut donc se contenter de donner un couple de coordonnées pour décrire un mouvement.

```
type mouvement = (int * int) * (int * int)
```

8. Écrire une fonction **mouvements : plateau -> mouvement list** qui renvoie la liste des mouvements possibles sur le plateau passé en paramètre.
9. Écrire une fonction **compte_pions : plateau -> int** qui renvoie le nombre de pions sur un plateau.
   En déduire une fonction **valide : plateau -> bool** qui indique si un plateau correspond à une partie gagnante.
10. Écrire deux fonctions **faire** et **defaire** de type **plateau -> mouvement -> unit** permettant de faire et défaire un mouvement.
11. Écrire une fonction **enumere : plateau -> mouvement list -> unit** telle que **enumere pos chemin** énumère les plateaux accessibles depuis **pos** jusqu'à obtenir une solution et sachant que **chemin** est la liste de mouvements, du plus récent au plus ancien, qui ont conduit jusqu'à **pos**.
    En cas de succès, on produira une exception **Solution of mouvement list** renvoyant la liste des mouvements ayant conduits à une solution.
    En déduire une fonction **resout : unit -> mouvement list** qui renvoie une liste de mouvements permettant de résoudre le solitaire.
    On prendra garde à renverser le chemin obtenu pour que le premier mouvement de la liste soit le premier mouvement effectué.

Amélioration : mauvaises positions

On se rend compte que de nombreuses positions sont réetudiées alors qu'on sait déjà qu'elles ne peuvent permettre d'aboutir à une solution. En effet, il y a souvent des coups indépendants pouvant être joués au même moment, ce qui fait qu'on peut aboutir à une même position de beaucoup de manière différente, ce qui augmente exponentiellement le nombre d'appels récursifs.

Une stratégie consiste a maintenir un ensemble de configurations mauvaises. Pour réaliser un tel ensemble, on va utiliser une table de hachage dont les clés sont les positions et les valeurs sont () (type unit). Si une position a une valeur associée dans la table, c'est qu'elle sera mauvaise.

Cela pose la question de la représentation persistante et immuable des positions. Une première stratégie peut consister à transformer le plateau en **case list list**. Cette stratégie est beaucoup trop coûteuse et elle ne permet pas de répondre instantanément.

On va profiter du fait qu'il n'y ait que 49 cases dans le plateau pour le représenter par un entiers sur 49 bits : $a_{00} + a_{10}2 + a_{20}2^2 + \cdots + a_{60}2^6 + a_{01}2^7 + \cdots + a_{66}2^48$ où a_{ij} vaut 1 lorsqu'il y a un pion sur la $j$ième ligne et la $i$ème colonne, c'est-à-dire quand **p.(i).(j) = Pion**.

Indication : l'einstruction **1 lsl n** s'évalue en 2^n. **lsl** signifie qu'on décale le chiffre 1 de n bits vers la droite dans son écriture binaire.

12. Écrire une fonction **code : plateau -> int** qui renvoie le numéro associé à un plateau.

Pour manipuler un ensemble, on va définir :

```
let mauvaises = Hashtbl.create 42
let ajoute x = Hashtbl.add mauvaises x ()
let contient x = Hashtbl.mem mauvaises x
```

L'appel à **ajoute x** rajoute **x** dans l'ensemble des mauvaises positions et **contient x** vérifie si **x** est dans cet ensemble.

13. Reprendre la fonction **enumere** avec un ensemble de mauvaises positions.