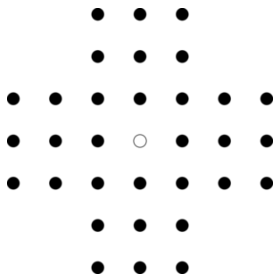
On considère ici le jeu du solitaire. On a un plateau comportant 33 emplacements et initialement 32 pions, représenté ici par des ronds noirs, et un emplacement vide au centre :



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

Exercice 1 (Première implémentation naïve) : On va représenter un plateau par le type OCaml suivant :

```
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 *)
```

1. Définir une fonction `print_plateau : plateau -> unit` qui affiche un plateau sous un format textuel lisible : on affichera un espace pour une case invalide, un . pour une case vide, et un # pour un pion.
2. 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 en fait se contenter de donner le couple de coordonnées pour décrire un mouvement.

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

3. Écrire une fonction `mouvements : plateau -> mouvement list` qui renvoie la liste des mouvements possibles sur le plateau passé en paramètres.
4. Écrire une fonction `compte_pions : plateau -> int` qui renvoie le nombre de pions sur un plateau.
5. En déduire une fonction `valide : plateau -> bool` qui indique si un plateau correspond à une partie gagnante.
6. Écrire deux fonctions `faire` et `defaire` de type `plateau -> mouvement -> unit` permettant de faire et défaire un mouvement.
7. Écrire une fonction `enumere : plateau -> mouvement list -> unit` telle que `enumere pos chemin` énumère les plateaux accessibles depuis `pos` jusqu'à obtenir une solution, 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 lèvera une exception `Solution of mouvement list` renvoyant la liste des mouvements ayant conduits à une solution.
8. En déduire une fonction `resout : unit -> mouvement list` qui revoie 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 à effectuer.

Ce code ne permet pas de calculer la solution, car il prend beaucoup trop de temps en raison du nombre de positions étudiées.

Exercice 2 (Cache des mauvaises positions) : On se rend compte que de nombreuses positions sont réétudiées alors qu'on sait déjà qu'elles ne peuvent pas 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 manières différentes, ce qui augmente exponentiellement le nombre d'appels récursifs.

Une stratégie consiste à 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 `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 couteuse, 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 entier 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 i-ème ligne et la j-ème colonne, i.e. quand `p.(i).(j) = Pion`.

9. Écrire une fonction `code : plateau -> int` qui renvoie le numéro associé à un plateau.
   Indication : l'entier `1 lsl n` vaut 2^n. `lsl` signifie qu'on décale le chiffre 1 de n bits vers la gauche dans son écriture binaire.

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.

10. Reprendre la fonction `enumere` avec un ensemble de mauvais codes.

Normalement, le code doit pouvoir permettre de réaliser la résolution en un temps raisonnable.

11. Écrire une fonction `affiche_solution : unit -> unit` qui anime la solution trouvée.
    On pourra utiliser l'instruction `Unix.sleep 1` qui permet d'attendre une seconde.