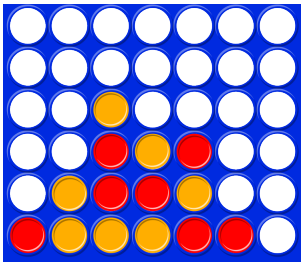
Le puissance 4 est un jeu d'alignement de pions. La situation initiale est une grille vide de 6 lignes par 7 colonnes. À son tour, un joueur place un jeton de sa couleur (rouge ou jaune) dans une colonne. Ce jeton tombe (par gravité) dans la première case non remplie de la colonne. Un joueur gagne la partie s'il aligne 4 jetons de sa couleur horizontalement, verticalement ou diagonalement.



```
type puissance4 = { grille : int array array ;
                    libres : int array ;
                    mutable joueur : int ;
                    mutable vides : int
                  }
```

On représente une partie de puissance 4 en OCaml par un objet du type `puissance4` ci-dessus, où :

- `grille` est une matrice de taille 6×7 contenant des 0, 1 ou 2, les 0 correspondant aux emplacements vides et les 1 et 2 aux jetons des deux joueurs ;
- `libres` est un tableau de taille 7 contenant le nombre d'emplacements libres dans chaque colonne de la grille ;
- `joueur` est un entier valant 1 ou 2 selon le joueur courant ;
- `vides` est un entier entre 0 et 42 correspondant au nombre de cases vides dans l'ensemble de la grille.

Le code ci-dessous représente un objet de type `puissance4` correspondant à la partie ci-dessus :

```
let p4 = {
    grille = [| [|0; 0; 0; 0; 0; 0; 0|];
                [|0; 0; 0; 0; 0; 0; 0|];
                [|0; 0; 1; 0; 0; 0; 0|];
                [|0; 0; 2; 1; 2; 0; 0|];
                [|0; 1; 2; 2; 1; 0; 0|];
                [|2; 1; 1; 1; 2; 2; 0|] |];
    libres = [|5; 4; 2; 3; 3; 5; 6|];
    joueur = 1;
    vides = 28 };;
```

Le fichier `TP09_eleve.ml` contient la définition du type et une fonction `afficher` permettant d'afficher une grille de puissance 4.

1 Fonctions utilitaires

Dans cette partie, on écrit des fonctions utilitaires pour gérer la partie.

1. Écrire une fonction `creer_p4 : unit -> puissance4` qui crée une partie de puissance 4 avec aucun coup joué.
2. Écrire une fonction `coup_possible : puissance4 -> int -> bool` qui prend en argument une partie et un entier et renvoie un booléen qui indique si un coup est possible en jouant dans la colonne correspondant à cet entier.
3. Écrire une fonction `coups_possibles : puissance4 -> int list` qui prend en argument une partie renvoie la liste des coups possibles (i.e. les numéros des colonnes qui ne sont pas pleines).
4. Écrire une fonction `jouer_coup : puissance4 -> int -> unit` qui joue un coup possible et fait toutes les modifications nécessaires.
5. Écrire de même une fonction `annuler_coup : puissance4 -> int -> unit` qui annule un coup joué.
6. Écrire une fonction `coup_gagnant : puissance4 -> int -> int -> bool` qui prend en argument une partie, un numéro de colonne et un numéro de joueur et indique si un coup dans la colonne pour le joueur est gagnant ou non pour ce joueur.

2 Stratégies

On cherche à créer des stratégies de jeu pour le puissance 4. Le nombre de grilles possibles étant de l'ordre de 5 mille milliards, une exploration exhaustive pourrait être envisageable, mais nécessiterait une certaine optimisation pour être réalisée en temps raisonnable.

7. Écrire une fonction `strategie_alea : puissance4 -> int` qui prend en argument un jeu de puissance 4 et renvoie un numéro de colonne choisi aléatoirement parmi les colonnes où un coup est possible. La fonction renverra -1 si toute la grille est remplie.
   Rappel : `Random.int n` renvoie un entier choisi aléatoirement et uniformément dans $[\![0, n-1]\!]$.
8. Écrire une fonction `comparer : (puissance4 -> int) -> (puissance4 -> int) -> int` qui prend en argument deux stratégies, simule une partie de puissance 4 et renvoie le numéro du gagnant (ou 0 si c'est un match nul). La fonction affichera l'état final de la grille avant de renvoyer l'entier.
9. Tester la fonction précédente en comparant deux stratégies aléatoires.

Pour mettre en place une stratégie Min-Max, on a besoin d'une heuristique pour évaluer une grille. On propose dans un premier temps l'heuristique suivante, qui consiste à attribuer un score à chaque pion posé, positivement pour le joueur 1 et négativement pour le joueur 2, selon la répartition ci-dessous :

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Ce tableau est fourni dans le fichier `TP09_eleve.ml`.
L'attribution des points part du constat qu'une position centrale permet plus facilement d'accéder à la victoire. On remarquera que `(n mod 2) - n/2` envoie les valeurs 0, 1 et 2 sur 0, 1 et -1 respectivement.

10. Écrire une fonction `heuristique : puissance4 -> int` qui calcule l'heuristique associée à une grille de puissance 4.
11. Écrire une fonction `minimax : (puissance4 -> int) -> int -> puissance4 -> int` qui prend en argument une fonction de calcul d'heuristique, un entier correspondant à une profondeur maximale et une partie de puissance 4 et renvoie le prochain coup à jouer selon l'algorithme du Min-Max. Si un coup gagnant est détecté pendant l'exploration, le score de la grille obtenue en jouant ce coup sera fixé égal à 1000 (ou -1000 pour le joueur 2).

Indication : on pourra écrire une fonction récursive auxiliaire qui renvoie un couple `(score, coup)` formé du meilleur score atteignable et du coup à jouer pour l'atteindre.

12. Comparer cette stratégie avec la stratégie aléatoire, pour différentes profondeurs maximales.

3 Améliorations

On souhaite améliorer les stratégies précédentes.

13. Écrire une fonction `alphabeta : (puissance4 -> int) -> int -> puissance4 -> int` qui prend en argument une fonction de calcul d'heuristique, un entier correspondant à une profondeur maximale et une partie de puissance 4 et renvoie le prochain coup à jouer selon l'algorithme du Min-Max **avec élagage Alpha-Beta**. On écrira une fonction auxiliaire qui prend en argument les bornes `alpha` et `beta` à utiliser pendant l'algorithme.
14. Comparer les deux stratégies précédentes, avec différents niveau de profondeur.

On propose de travailler avec une autre heuristique : pour chaque bloc de quatre cases alignées, on attribue un score à ce bloc :

- si chaque case contient un pion de la même couleur, on donne le score $+1000$ ou -1000 selon la couleur ;
- si ce bloc est vide ou contient des pions de couleurs différentes, on donne le score 0 (aucun alignement gagnant ne peut se faire dans ce bloc) ;
- sinon, on attribue le score ± 10, ± 3 ou ± 1 selon qu'il y a 3, 2 ou 1 pion d'une couleur (positif ou négatif selon sa couleur).

L'heuristique consiste alors à additionner tous les scores des blocs de quatre.

15. Écrire une fonction `heuristique2 : puissance4 -> int` qui calcule cette heuristique.

4 Jouer contre la machine

Pour finir, on souhaite implémenter un moyen de faire jouer l'utilisateur. Pour cela, on va implémenter une "stratégie" qui demande quel coup jouer à l'utilisateur (qui devra alors taper son choix dans la console).

16. Écrire une fonction `comparer_anim : (puissance4 -> int) -> (puissance4 -> int) -> int` faisant la même chose que la fonction `comparer` mais affichant la grille à chaque étape.
    On pourra utiliser l'instruction `Unix.sleep 1` pour éviter que tout s'affiche d'un coup.
17. Écrire une fonction `humain : puissance4 -> int` qui demande à l'utilisateur quel coup jouer (et le renvoie).
    On pourra utiliser la fonction `read_int : unit -> int` qui lit un entier sur l'entrée standard.
18. Pouvez-vous battre les heuristiques précédentes ?