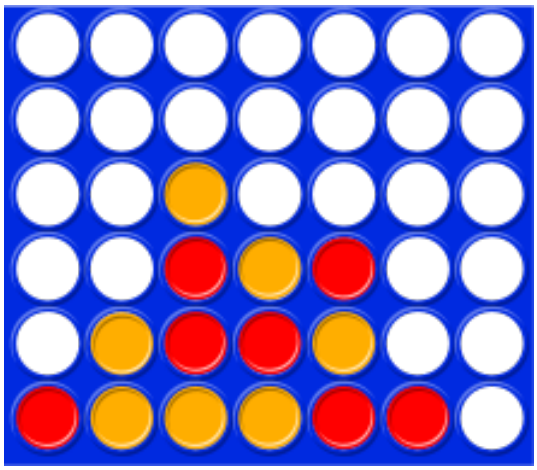
I Présentation du problème

Lobjectif de ce TP est de numériser puis jouer informatiquement au jeu du puissance 4



Dans ce jeu, chacun des deux joueurs (noté J_0 et J_1 dans la suite) joue a tour de rôle en plaçant une pastille colorée dans une colonne. Celle-ci tombe au plus bas. Le but est de réussir à aligner (dans une ligne, colonne, ou diagonale) 4 pastilles de la même couleur.

Numérisation

On convient de représenter un état de la grille de jeu par une liste de listes.
Chaque liste représente une ligne de la grille de jeu et chaque élément d'une ligne représente une case.
Par convention, les éléments de chaque ligne seront les entiers 0 ou 1 ou `None`. Un entier i représente le fait que le joueur J_i a positionné une pastille à cet endroit et la valeur `None` représente les cases vides.
Ainsi, si on admet que J_0 joue en rouge et J_1 en jaune, l'état de la grille d'exemple précédente est numérisé par

```
G = [
    [0, 1, 1, 1, 0, 0, None],
    [None, 1, 0, 0, 1, None, None],
    [None, None, 0, 1, 0, None, None],
    [None, None, 1, None, None, None, None],
    [None, None, 0, None, None, None, None],
    [None, None, 0, None, None, None, None]
]
```

La ligne du bas est la première ligne stockée (une pastille ira à l'indice le plus petit possible). Remarquez les crochets extérieurs.
Dans la suite nous nommerons *situation* un couple

```
s = (joueur, G)
```

où `joueur` est l'entier désignant le prochain joueur à jouer et `G` la grille de l'état actuel.

Les fonctions fournies

Plusieurs fonctions sont déjà pré-remplies dans la feuille de script de ce TP.
Vous pouvez lire le code des fonctions présentes mais non mentionnées ici, mais vous ne devriez pas avoir à vous en servir vous même.

- `affiche(s)` qui prend comme argument une situation et affiche dans la console une représentation de la grille.
  La convention utilisée est de représenter le joueur 0 par des "0" et le joueur 1 par des "X".
- `jeu_a_deux` qui ne prend pas d'argument et qui permet de simuler une partie entre 2 humains dans la console. Pour l'instant plusieurs pièces manquent à l'édifice pour que cette fonction puisse faire son office.
- `jouer_IA_nulle` qui ne prend pas d'argument et permet de jouer contre l'ordinateur prenant des décisions prévisibles (voir plus loin)
- `jouer` qui a deux arguments optionnels permettant de régler la quantité de calcul que fera le programme avant de jouer (voir plus loin).

II Jouer à 2

Nous avons besoin de plusieurs outils pour pouvoir faire avancer le jeu pas à pas.

Exercice 1
Compléter la fonction `depose_pion` qui permet d'obtenir la situation après un coup joué.

Pour tester si une variable `var` contient la valeur `None` on utilise la syntaxe suivante (rappelons que le `else` est optionnel en python)

```
if var is None:
    # exécuté si var contient None
# ou alors
if var is not None:
```

Nous allons maintenant déterminer si une partie est finie après examen d'un coup joué.

Exercice 2

1. A quelle condition une partie peut-elle être déclarée match nul ?
2. Compléter la fonction `partie_nulle` qui prend comme argument une situation.
3. On doit maintenant vérifier si un coup a amené à une situation gagnante.
   On se place dans la situation ou la grille vient d'être modifiée par un joueur qui a joué le coup (i, j) (ligne i, colonne j)
   (a) Combien de listes de 4 positions adjacentes dans la ligne i devons-nous vérifier a priori (indépendamment de la valeur de j)
   (b) Même question pour la colonne j puis pour les deux diagonales.
   (c) Compléter la liste `a_verifier` dans la fonction `gagnant` avec toutes les listes de longueur 3 correspondant aux cases qui devraient avoir la même valeur que la case (i, j) pour obtenir une position gagnante.
   (d) Finir de compléter la fonction `gagnant`

Exercice 3
Notre dernier outil est la fonction `eval_f`.
Pour définir les valeurs de retour de cette fonction (et être cohérent avec la suite de ce TP), on utilise la valeur spéciale `inf` qui a été importée du module `math`. Il s'agit d'un nombre plus grand que tout entier et tout flottant. Son opposé est `-inf`.

Exercice 4
Si tout va bien, la fonction `jeu_a_deux` devrait maintenant fonctionner et vous pouvez en profiter !

III Jouer contre une IA

Nous allons maintenant mettre en œuvre des algorithmes permettant à notre programme de choisir un coup à jouer en fonction de l'état actuel de la grille.

Graphe

Le déroulement d'une partie de puissance 4 (mais pas seulement ce jeu) peut-être décrit à l'aide du graphe (orienté) des états possibles.

- Un nœud est une situation (joueur, grille)
- Une arête est un coup possible.

Ainsi de chaque nœud partent tous les coups possibles et une partie est un fait un chemin dans ce graphe.

Exercice 5
Expliquer rapidement pourquoi le graphe précédent ne comporte pas de cycle pour le jeu du puissance 4. Est-ce le cas pour les échecs ?

Construction et parcours

Exercice 6

1. Compléter la fonction `fils` qui étant donné une situation retourne la liste de toutes les situations atteignables.

   Si b est un bit qui vaut 0 ou 1, alors l'autre bit l'obtient par $1 - b$

   Cette fonction permet de construire le graphe complet pas à pas. Mais....
2. À partir de la situation de départ `(1, grille0)` (la variable est définie au début du code) combien de situations sont atteignables au bout de 1 coup ? Au bout de 2 ?
   On trouve 2401 nœud pour le 4ème coup. Proposer une formule donnant le nombre de situations au k-ième coup et donner son domaine de validité.

En pratique on ne peut pas construire le graphe complet pour tenter de choisir un coup qui conduirait à la victoire.

Heuristique et évaluation des positions

Une stratégie classique, pour éviter l'explosion combinatoire évoquée, est de ne construire qu'une partie du graphe.

- on fixe k et se on limite à regarder les k prochains coups.
- Pour chaque position atteinte, on calcule un nombre représentant à quel point la position est favorable à chaque joueur en utilisant une approximation du jeu (cette approximation est appelée une heuristique).
- Si on trouve une position gagnante pour J_0 la valeur ne sera pas une approximation et notre convention sera que cette valeur est `inf` (le nombre le plus positif possible). De même la valeur d'une position gagnante pour J_1 sera `-inf`.
- Dans tous les cas, notre programme va choisir (comme expliqué plus loin) le coup qui conduit à la position ayant le score maximum (on fait jouer l'ordinateur en comme J_0).

Dans le cadre de ce TP, une **heuristique** est une fonction qui prend comme argument une situation et doit retourner un nombre ou `inf` ou `-inf`.

Exercice 7

1. La première heuristique est gracieusement fournie dans le code, il s'agit de `h_constante`. Comment l'interpréter ?
   Faire une partie en exécutant la commande `jouer_IA_nulle()` dans la console pour valider votre analyse.
2. La deuxième stratégie que nous allons essayer est un peu plus efficace.
   Notre heuristique va, à partir d'une situation donnée, finir la partie en jouant aléatoirement pour chaque joueur et décider qui gagne. On répète ce procédé un nombre fixé de fois et le score calculé est alors le nombre de victoires de J_0 - le nombre de victoires de J_1 (pour être cohérent avec notre convention).
   (a) Compléter la fonction `fin_alea` qui complète une partie de manière aléatoire (son argument est une situation). Remarquez que la convention de valeur de retour est cohérente avec la fonction `eval_f`

Pour gérer l'aléatoire, on pourra utiliser `random.randint` ou `random.choice`

   (b) Dans le corps de la fonction `h_alea`, compléter la fonction `h` qui calcule le score évoqué en jouant `nb` parties
3. Nous pouvons maintenant comprendre le sens des arguments optionnels de la fonction `jouer` qui servent à régler la difficulté de notre IA.
   Plus k est grand et plus le nombre de partie aléatoires considérées est grand, meilleur sera le choix de notre algorithme. Le réglage par défaut (30 parties, $k = 3$) est difficile à battre ! Testez

Pour aller plus loin

Le cœur de notre algorithme de décision n'a pas été explicité ici. La ligne magique se situe à

```
import mystere
minimax_h = mystere.cre_minimax_h(eval_f, fils, construit_cle)
```

Dans le module `mystere`, on a implémenté l'algorithme minimax utilisant une heuristique. Votre but est de ne pas aller voir le code mais de ré-implémenter cet algorithme.
Il s'agit d'un parcours récursif du graphe des états en se limitant à k coups. Face à une situation :

- Si la situation est finale (gagnante ou nulle), on peut conclure.
- Si le nombre de coup maximal est atteint on utilise l'heuristique
- Sinon, on calcule récursivement toute les valeurs pour tous les coups possibles et, suivant qui jouait, on retourne la valeur maximale ou minimale trouvée (évidemment, si c'était au joueur J_1 de jouer, on lui choisit la position la moins favorable).

Pour éviter de calculer de nombreuses fois la même quantité, on se sert des techniques vues en programmation dynamique : on retient dans des dictionnaires les informations déjà calculées.
Les dictionnaires utilisent une situation comme clé (d'où la fonction `construit_cle` qui transforme une situation en clé utilisable dans un dictionnaire). On stocke dans 3 dictionnaires les évaluations, les coups et les positions obtenues.
La valeur de retour doit être un couple (score, situation).

Exercice 8
Créer la fonction `minimax_h2` qui doit remplacer la fonction `minimax_h`. Les arguments doivent être, dans l'ordre : `s, coup, d_eval, d_stra, d_coup, h, k` où `d_stra` est le dictionnaire des situations, `h` la fonction d'heuristique et les autres arguments sont transparents à ce stade du TP.

Exercice 9
En vous inspirant du code de `jouer`, comparer les performances de deux réglages de notre IA en les faisant jouer l'une contre l'autre. Peut-on compenser une valeur plus petite de k par une valeur plus grande de nb ?
On pourra utiliser

```
jeu(1, grille0, IA, IA2, affiche_etat=False)
```

qui passe à l'argument optionnel `affiche_etat` la valeur `False` et évite tous les `print`.