p4-v2

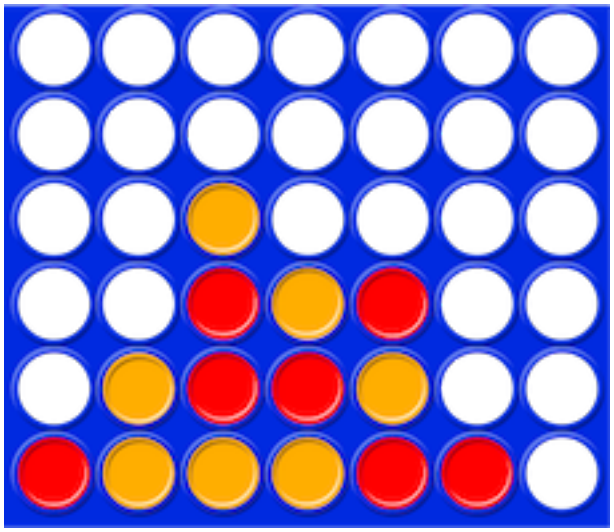
March 12, 2024

1 Présentation

Ce sujet propose de développer un jeu *puissance 4*. Les questions vous guident dans ce développement. Des tests d'auto-validation suivent chaque question et vous aident à valider vos traitements. **Ne pas continuer dès qu'un test déclenche une exception ou produit des messages d'erreur.** Dans ce cas, vos traitements sont erronés et vous devez les corriger avant d'aller plus loin dans la résolution du sujet.

1.1 Le jeu puissance 4

Le jeu **puissance 4** est un jeu de stratégie dont le but est d'aligner une suite de 4 pions de même couleur sur une grille comptant 6 lignes horizontales et 7 colonnes verticales.



Il y a deux joueurs. Chacun dispose de 21 pions d'une couleur, en général jaune ou rouge.

Tour à tour, les joueurs placent un pion dans la colonne de leur choix. **Le pion tombe dans la colonne jusqu'à la position la plus basse possible**. Puis c'est à l'adversaire de jouer de façon similaire avec ses jetons.

Le vainqueur est le joueur qui réalise le premier un **alignement horizontal ou vertical ou diagonal d'au moins quatre pions consécutifs de sa couleur**.

Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.

1.2 La grille et les jetons de couleurs

Constantes et `couleurs`.

- `NBL` et `NBC` sont des constantes entières égales à 6 et 7 respectivement. Elles représentent le nombre de lignes et le nombre de colonnes du jeu réel.
- Les couleurs jaune et rouge sont **ici** des constantes entières égales à 1 et 2 respectivement.
- L'énumératif `couleurs` permet de définir le type de paramètre(s) de couleur dans la définition de fonctions.
- Il est complété par la constante 3 qui représente un jeton de couleur quelconque (jaune ou rouge).

[ ]:
```
NBL = 6 # nbre de lignes
NBC = 7 # nbre de colonnes

couleurs = (1, 2, 3) # (jaune, rouge, jaune ou rouge)
```

La grille de jeu et le tableau qui la représente.

La grille du jeu est un tableau 2D de valeurs entières.

IMPORTANT : ce tableau stocke la grille de jeu ligne par ligne **à partir de la ligne du bas de la grille** - la ligne 0 du tableau représente la ligne du bas de la grille de jeu, cad. la ligne qui accueille le premier jeton joué dans une partie, - la ligne 1 du tableau représente la ligne de la grille qui accueille le deuxième jeton joué si celui-ci est glissé dans la même colonne que le premier jeton, et ainsi de suite. - Il faudra être prudent **dès que** les notions de "haut" et de "bas" de la grille de jeu seront considérés ; - en particulier, les `print()` du tableau affichent en fait la grille "tête en bas".

Dans la suite, la grille de jeu sera de taille `NBL x NBC`. Cependant les fonctions qui définissent les traitements de la grille de jeu seront paramétrés par `dimv` et `dimh` pour représenter des grilles de taille quelconque `dimv x dimh`.

La cellule suivante définit deux états de la grille de jeu. - **la valeur 0 représente une case vide** - la grille `g0` est vide – comme en début de jeu. Elle est représentée comme un tableau 2D de zéros - la grille `g1` correspond à l'image du début du sujet

[ ]:
```
g0 = [[0 for j in range(NBC)] for i in range(NBL)]

g1 = [[2,1,1,1,2,2,0],
      [0,1,2,2,1,0,0],
      [0,0,2,1,2,0,0],
      [0,0,1,0,0,0,0],
      [0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0]]

print(g0)
print(g1)
```

Remarque sur l'affichage de la grille.

La grille de jeu étant représentée par un tableau 2D, on peut utiliser la fonction d'affichage `print()` appliquée à un tableau. Bien sûr, on gardera bien en tête que l'affichage alors obtenu correspond **aux lignes à partir du bas de la grille**.

Exemple. L'exécution de

`print(g1)`

pour la grille `g1`de la figure du début donne :

```
[[2, 1, 1, 1, 2, 2, 0], [0, 1, 2, 2, 1, 0, 0], [0, 0, 2, 1, 2, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0
```

($\star$) Cet affichage sera modifié en une version plus réaliste en fin de sujet.

2 Quelques primitives utiles

On va écrire deux fonctions qui retournent le nombre de jetons d'une grille donnée :

1. `nbJetons0()` qui se limite aux jetons jaunes ou rouges,
2. `nbJetons()` qui compte tous les jetons indépendamment de leur couleur.

2.1 `nbJetons0()`: jaunes ou rouges

Ecrire selon l'en-tête suivant, la fonction nbJetons0() qui retourne le nombre de jetons de couleur `c` d'une grille `g` de taille `dimv x dimh`. Le paramètre `c` prendra les valeurs 1 ou 2 dans cette première version.

[ ]:
```
def nbJetons0(c : couleurs, g : list[list[int]], dimv : int, dimh : int) -> int:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

2.1.1 Auto-validation

[ ]:
```
assert nbJetons0(1, g0, NBL, NBC) == 0
assert nbJetons0(1, g1, NBL, NBC) == 7
```

2.2 `nbJetons()`: jaunes, rouges, jaune ou rouge

Reprendre `nbJetons0()` et la compléter de façon à ce que le paramètre couleur `c` prenne aussi la valeur `3` et retourne dans ce cas tous les jetons (jaunes ou rouges) présents dans la grille de jeu.

[ ]:
```
def nbJetons(c : couleurs, g : list[list[int]], dimv : int, dimh : int) -> int:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

2.2.1 Auto-validation

```
[ ]: assert nbJetons(1, g0, NBL, NBC) == 0
     assert nbJetons(3, g0, NBL, NBC) == 0
     assert nbJetons(2, g1, NBL, NBC) == 7
     assert nbJetons(3, g1, NBL, NBC) == 14
```

2.3 `nbCasesLibres()`

Ecrire selon l'en-tête suivant, une fonction NbCasesLibres() qui retourne le nombre de cases vides (sans jeton) d'une grille de taille `dimv x dimh`.

```
[ ]: def NbCasesLibres(g : list[list[int]], dimv : int, dimh : int) -> int:
     # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
     raise NotImplementedError()
```

```
[ ]: # Ne pas écrire dans cette cellule
```

2.3.1 Auto-validation

```
[ ]: assert NbCasesLibres(g1, NBL, NBC) == 28
```

2.4 `estValide()`

Cette question n'est pas indispensable pour la suite des traitements.

Une grille est valide si les conditions suivantes sont satisfaites.
- la grille comporte 6 lignes et 7 colonnes
- les pions jaunes commençant, il y a le même nombre de pions jaunes et de pions rouges ou un pion jaune de plus
- il y a 21 pions jaunes au plus
- il n'y a pas de case vide "sous" une case non vide

Ecrire une fonction `estValide()` selon l'en-tête suivante qui vérifie la validité d'une grille.

```
[ ]: def estValide(g : list[list[int]], dimv : int, dimh : int) -> bool:
     # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
     raise NotImplementedError()
```

```
[ ]: # Ne pas écrire dans cette cellule
```

2.4.1 Auto-validation

Validez la cellule suivante qui défini la grille **non-valide** `gbad` utilisée pour les tests et les auto-validations.

```
[ ]: g1 = [[2, 1, 1, 1, 2, 2, 0], [0, 1, 2, 2, 1, 0, 0], [0, 0, 2, 1, 2, 0, 0], [0,␣
       ↪0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
     gbad = [[2, 1, 1, 1, 2, 2, 0], [0, 1, 2, 2, 1, 0, 0], [0, 0, 0, 1, 2, 0, 0],␣
       ↪[0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
```

La cellule d'auto-validation suivante doit s'exécuter sans erreur.

```
[ ]: assert estValide(g1, NBL, NBC) == True
     assert estValide(gbad, NBL, NBC) == False
```

3 Alignements gagnants

On va examiner les 4 alignements gagnants d'une couleur donnée. - alignement horizontal : 4 pions de même couleur consécutivement alignés sur une ligne
- alignement vertical : 4 pions de même couleur consécutivement alignés sur une colonne
- alignement selon une diagonale NordOuest-SudEst *du tableau* : 4 pions de même couleur consécutivement alignés sur une diagonale "du haut à gauche vers le bas à droite"
- alignement selon une diagonale NordEst-SudOuest *du tableau* : 4 pions de même couleur consécutivement alignés sur une diagonale "d'en haut à droite vers le bas à gauche"

Attention. Les 2 derniers alignements dépendent du haut et du bas **du tableau** qui représente la grille :

- la ligne 0 est le haut du tableau (qui représente le bas de la grille)
- la ligne `NBL - 1` est le bas du tableau (qui représente le haut de la grille)

3.1 `horiz()`

Ecrire selon l'en-tête suivant, la fonction `horiz()` qui identifie un alignement horizontal gagnant d'une couleur donnée : 4 pions de même couleurs consécutivement alignés sur une ligne.

Cette fonction retourne le numéro de la ligne de l'alignement trouvé ou -1 sinon.

```
[ ]: def horiz(c : couleurs, g : list[list[int]],  dimv : int, dimh : int) -> int:
     # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
     raise NotImplementedError()
```

```
[ ]: # Ne pas écrire dans cette cellule
```

3.1.1 Auto-validation

L'auto-validation suivante doit s'exécuter sans erreur.

```
[ ]: g0 = [[0 for j in range(NBC)] for i in range(NBL)]
     g0[0] = [1, 2, 1, 2, 1, 2, 1]
     g0[1] = [1, 1, 2, 1, 1, 2, 1]
     g0[2] = [1, 2, 2, 2, 2, 2, 0]

     assert horiz(1, g0, NBL, NBC) == -1
     assert horiz(2, g0, NBL, NBC) == 2
```

3.2 `vertic()`

Ecrire selon l'en-tête suivant, la fonction `vertic()` qui identifie un alignement vertical gagnant d'une couleur donnée (4 pions de même couleur consécutivement alignés sur une colonne).

Cette fonction retourne le numéro de la colonne de l'alignement trouvé ou -1 sinon.

[ ]:
```
def vertic(c : couleurs, g : list[list[int]],  dimv : int, dimh : int) -> int:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

3.2.1 Auto-validation

On définit deux grilles **non valides** mais utiles pour vérifier l'alignement vertical gagnant pour chacune des couleurs.

L'auto-validation suivante doit s'exécuter sans erreur.

[ ]:
```
g_testvert = [[0 for j in range(NBC)] for i in range(NBL)]
for i in range(NBL-2):
    g_testvert[i][3] = 1
for i in range(NBL-3):
    g_testvert[i][NBC-1] = 2

assert vertic(1, g_testvert, NBL, NBC) == 3
assert vertic(2, g_testvert, NBL, NBC) == -1

g_testvert[2][3] = 2
g_testvert[3][NBC-1] = 2

assert vertic(1, g_testvert, NBL, NBC) == -1
assert vertic(2, g_testvert, NBL, NBC) == 6
```

3.3 `diagoNOSE()`

Ecrire selon l'en-tête suivant, la fonction `diagoNOSE()` qui identifie un alignement diagonal gagnant d'une couleur donnée : 4 pions consécutifs de même couleur diagonalement alignés.

Cette fonction retourne 1 si un tel alignement est présent ou -1 sinon.

Rappel.

- la ligne 0 est le haut du tableau (qui représente le bas de la grille)
- la ligne `NBL-1` est le bas du tableau (qui représente le bas de la grille)
- diagonale NordOuest-SudEst : "du haut à gauche vers le bas à droite" *du tableau*

[ ]:
```
def diagoNOSE(c : couleurs, g : list[list[int]],  dimv : int, dimh : int) ->␣
↪int:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

3.3.1 Auto-validation

L'auto-validation suivante doit s'exécuter sans erreur.

[ ]:
```
g_testdiago1 = [[0 for j in range(NBC)] for i in range(NBL)]
for i in range(4):
    for j in range(2, 6):
        g_testdiago1[i][j] = 2

assert diagoNOSE(1, g_testdiago1, NBL, NBC) == -1
assert diagoNOSE(2, g_testdiago1, NBL, NBC) == 1

g_testdiago1[0][2] = 1
assert diagoNOSE(1, g_testdiago1, NBL, NBC) == -1
assert diagoNOSE(2, g_testdiago1, NBL, NBC) == -1

g_testdiago1[4][5
] = 2
assert diagoNOSE(1, g_testdiago1, NBL, NBC) == -1
assert diagoNOSE(2, g_testdiago1, NBL, NBC) == 1
```

3.4 `diagoNESO()`

Ecrire selon l'en-tête suivant, la fonction `diagoNESO()` qui identifie l'autre alignement diagonal gagnant d'une couleur donnée.
Cette fonction retourne 1 si un tel alignement est présent, ou -1 sinon.

Rappel.

- la ligne 0 est le haut du tableau (qui représente le bas de la grille)
- la ligne `NBL-1` est le bas du tableau (qui représente le bas de la grille)
- diagonale NordEst-SudOuest : diagonale "d'en haut à droite vers le bas à gauche" du tableau

[ ]:
```
def diagoNESO(c : couleurs, g : list[list[int]],  dimv : int, dimh : int) ->␣
 ↪int:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

3.4.1 Auto-validation

L'auto-validation suivante doit s'exécuter sans erreur.

[ ]:
```
g_testdiago1 = [[0 for j in range(NBC)] for i in range(NBL)]
for i in range(4):
    for j in range(2, 6):
        g_testdiago1[i][j] = 2

assert diagoNESO(1, g_testdiago1, NBL, NBC) == -1
```

```
assert diagoNESO(2, g_testdiago1, NBL, NBC) == 1

g_testdiago1[0][5] = 1
assert diagoNESO(1, g_testdiago1, NBL, NBC) == -1
assert diagoNESO(2, g_testdiago1, NBL, NBC) == -1

g_testdiago1[4][2] = 2
assert diagoNESO(1, g_testdiago1, NBL, NBC) == -1
assert diagoNESO(2, g_testdiago1, NBL, NBC) == 1
```

3.5 `est_gagnante()`

Ecrire selon l'en-tête suivant, la fonction `est_gagnante()` qui vérifie si la grille `g` est gagnante pour la couleur `c`.

Cette fonction retourne un booléen.

[ ]:
```
def est_gagnante(c : couleurs, g : list[list[int]], dimv : int, dimh : int) ->␣
↪bool:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

3.5.1 Auto-validation

L'auto-validation suivante doit s'exécuter sans erreur.

[ ]:
```
g0 = [[0 for j in range(NBC)] for i in range(NBL)]
faux = est_gagnante(1, g0, NBL, NBC)
assert faux == False
faux = est_gagnante(1, g0, NBL, NBC)
assert faux == False

for i in range(0,4):
    g0[i][3] = 1
for j in range(0,3):
    g0[0][j] = 2
g0[0][4] = 2

vrai = est_gagnante(1, g0, NBL, NBC)
assert vrai == True
faux = est_gagnante(2, g0, NBL, NBC)
assert faux == False
```

3.6 `jouer()` !

On commence avec une fonction qui permet l'interaction entre un joueur et le jeu.

Ecrire selon l'en-tête suivant, la fonction `jouer()` qui ajoute un pion de couleur `c` à la colonne `colonne` à la grille de jeu en cours `g` (de dimensions habituelles).

- Les numéro des colonnes de la grille de jeu `g` et du tableau qui la représente sont égales.
- Cette fonction modifie l'état de la grille `g`. C'est donc une procédure qui ne retourne aucun résultat (autre que `None` en python).

Pour cette version simple de `jouer()`, on supposera que les valeurs de `c` et de `colonne` ont du sens et ne rendent pas la grille non valide.

[ ]:
```
def jouer(c : couleurs, colonne : int, g : list[list[int]], dimv : int, dimh :␣
 ↪int) -> None:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

3.6.1 Auto-validation

L'auto-validation suivante doit s'exécuter sans erreur.

[ ]:
```
# g1 dans son état initial
g1 = [[2, 1, 1, 1, 2, 2, 0], [0, 1, 2, 2, 1, 0, 0], [0, 0, 2, 1, 2, 0, 0], [0,␣
 ↪0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]

# un tour de jeu
jouer(1, 1, g1, NBL, NBC)
jouer(2, 3, g1, NBL, NBC)

# vérifications
assert g1[2][1] == 1
assert g1[3][3] == 2
```

3.7 (⋆) Affichages de la grille

Objectif 20. Un affichage plus agréable va être développé en 2 étapes :
- d'abord un affichage de la grille uniquement, - puis un affichage complété avec les numéros des lignes et des colonnes tels que les voient les joueurs.

3.7.1 (⋆) `grille_en_str()`

Le premier affichage sera tel que : - les cases vides sont affichées avec le caractère `'.'` - les pions jaunes sont affichées avec le caractère `'O'`
- les pions rouge sont affichées avec le caractère `'X'`

Pour cela, on va écrire une fonction `grille_en_str( )` qui transforme la grille de jeu en une chaîne de caractères adaptée qui, ensuite, pourra être affichée d'un seul `print` et donner l'affichage souhaité.

- L'en-tête de cette fonction est définie dans la cellule suivante.
- Cette fonction retourne la chaîne de caractère construite par l'algorithme suivant :

- chaque ligne de la grille est interprétée comme une chaîne de 6 caractères séparés par des espaces ;
- chaque chaîne de caractères est concaténée dans un ordre adéquat dans une chaîne globale qui représentera la grille complète ;
- cette chaîne globale sera retournée pour obtenir l'affichage demandé.

On rappelle que le *caractère* spécial `\n` permet un saut de ligne.

Ainsi

```
print( grille_en_str(g1) )
```

donnera l'affichage de la grille `g1` suivant.

```
. . . . . . .
. . . . . . .
. . O . . . .`
. O X O X . .
. O X X O . .
X O O O X X .
```

[ ]:
```
def grille_en_str(g : list[list[int]], dimv : int, dimh : int) -> str:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

(⋆) **application**

Vérifier que l'affichage suivant de `g1` est identique à celui indiqué au dessus.

[ ]:
```
print(grille_en_str(g1, NBL, NBC))
```

3.7.2 (⋆) `afficher()`

Attention : bien lire ce passage qui définit 2 numérotations différentes.

Jusqu'à présent, on utilise une numérotation induite par le tableau qui représente la grille de jeu : la *numérotation-développeur*. `print()` permet un affichage selon cette numérotation.

On définit maintenant une *numérotation-joueur* : le *joueur* compte

- les lignes à partir de 1 et en partant de la ligne du **bas** de la grille du jeu **réel**
- les colonnes à partir de 1 et en partant de la colonne de gauche de la grille (voir exemple ci-dessous).

Par la suite, nous indiquerons **explicitement** les cas où cette numérotation-joueur devra être utilisée (pour `afficher()`, `atoidejouer()` et `jeu()`). Sinon, la numérotation-développeur reste celle à utiliser.

Si besoin : - `afficher()` est adapté pour visualiser la *numérotation-joueur* - `print()` (du tableau) est adapté pour visualiser la *numérotation-développeur*.

Ecrire la fonction `afficher()` selon l'en-tête suivant où le paramètre booléen `debug` permet de compléter l'affichage précédent avec les numéros-joueur des lignes et des colonnes de la grille comme ci-dessous pour `g1`.

```
. . . . . . . |6
. . . . . . . |5
. . O . . . . |4
. . X O X . . |3
. O X X O . . |2
X O O O X X . |1
-------------
1 2 3 4 5 6 7
```

Si `debug == True`, `afficher()` réalisera l'affichage ci-dessus.

Si `debug == False`, `afficher()` réalisera l'affichage sans numéro de lignes, ni de colonnes.

Cette fonction pourra s'inspirer de la fonction `grille_en_str()`. Elle réalise l'affichage demandé : il s'agit donc d'une *procédure* d'affichage (qui mélange des traitements spécifiques et des entrées-sorties).

[ ]:
```
def afficher(g : list[list[int]], dimv : int, dimh : int, debug : bool) -> None:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

[ ]:
```
# Ne pas écrire dans cette cellule
```

(⋆) **application**

Vérifier que les affichages de `g1` sont identiques à ceux indiqués au dessus.

[ ]:
```
print(afficher(g1, NBL, NBC, debug=False))
print(afficher(g1, NBL, NBC, debug=True))
```

3.8 `atoidejouer()`

Ecrire selon l'en-tête suivant, une fonction `atoidejouer()` qui demande au joueur *d'entrer au clavier* dans quelle colonne de la **grille réelle** `g` il dépose son jeton de couleur `c`.

Attention : numérotation. Cette fonction (d'ES)

- prend "en entrée" le **numéro-joueur** de colonne saisi au clavier,
- retourne le **numéro-développeur** de la colonne correspondant pour les traitements à suivre.

[ ]:
```
def atoidejouer(c : couleurs, g : list[list[int]], dimv : int, dimh : int) ->␣
↪int:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

3.8.1 Auto-validation

Entrez successivement `1` et `2` et l'auto-validation suivante doit s'exécuter sans erreur.

[ ]:
```
g0 = [[0 for j in range(NBC)] for i in range(NBL)]

zero = atoidejouer(1, g0, NBL, NBC)
assert zero == 0

un = atoidejouer(2, g0, NBL, NBC)
assert un == 1
```

3.9 jeu()

On peut maintenant utiliser les développements précédents pour écrire selon l'en-tête suivant, une fonction `jeu()` qui permet à deux joueurs de jouer comme suit.
- Avec `atoidejouer()`, demander à chaque joueur de jouer à son tour, en commençant par les jaunes. - On suppose (ici) que le joueur choisit une colonne valide - La grille de jeu est affichée après chaque action de jeu
- Le jeu s'arrête quand un des joueurs a gagné ou en cas d'égalité (blocage) - Le résultat est **affiché à l'écran** comme suit :

```
Les pions jaunes/rouges gagnent. Bravo !
```

ou en cas d'égalité (blocage) :

```
Egalité ... Recommencez !
```

La partie s'effectue en complétant la grille `g` qui est : - soit une grille vide à créer avant l'appel, - soit une grille partiellement remplie provenant de la lecture d'un fichier (comme G1.txt). Dans ce cas, il faut donc vérifier la validité de la grille lue dans le fichier et, si besoin afficher le message :

```
Grille invalide. Jeu stoppé.
```

[ ]:
```
def jeu(g : list[list[int]], dimv : int, dimh : int) -> None:
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

3.9.1 Auto-validation 1

Entrez successivement `2, 2` pour que l'exécution de la cellule suivante fasse gagner les rouges (`X`).

[ ]:
```
# g1 dans son état initial
g1 = [[2, 1, 1, 1, 2, 2, 0], [0, 1, 2, 2, 1, 0, 0], [0, 0, 2, 1, 2, 0, 0], [0,␣
 ↪0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
print("Entrer 2 puis 2 et les rouges gagnent")
jeu(g1, NBL, NBC)
```

3.9.2 Auto-validation 2

Entrez successivement `2, 1, 2` pour que l'exécution de la cellule suivante fasse gagner les jaunes (`O`).

[ ]:
```

```

```
g1 = [[2, 1, 1, 1, 2, 2, 0], [0, 1, 2, 2, 1, 0, 0], [0, 0, 2, 1, 2, 0, 0], [0,␣
  ↪0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
print("Entrer 212 et les jaunes gagnent")
jeu(g1, NBL, NBC)
```

[ ]: