cc-2024-03-22-v010

March 22, 2024

1 Contrôle continu du 22 mars 2024

Les fonctions demandées sont décrites partiellement : signature, documentation (doc-string) et pré-conditions.
Ne pas modifier ces éléments qui sont donnés pour vous aider.
En revanche, vos réponses doivent remplacer les 2 dernières lignes de ces cellules :

```
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

Bon travail.

1.1 Exercice 1 : une fonction inconnue

1.1.1 Quoi

Que calcule et retourne la fonction `f` ?

[ ]:
```
def f(l: list[int], n: int, s: int) -> int:
    '''Que fait f ?'''
    assert n == len(l)
    assert n > 0
    c = 0
    for i in range(len(l)):
        if l[i] < s :
            c = c + 1
    return c
```

VOTRE REPONSE CI DESSOUS

1.1.2 C'est sûr ?

Six premiers tests unitaires Utiliser les 3 listes d'entiers `[1]`, `[1, 1]` et `[1, 2]`, pour écrire 6 tests unitaires simples de cette fonction `f`.

[ ]:
```
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

D'autres tests unitaires Compléter les tests suivants sur la liste `l0` suivante.

[ ]:
```python
l0 = [i + 2*(i%2) for i in range(0, 10)]
print(l0)

assert f(l0, len(l0), 0) == .
assert f(l0, len(l0), 1) == .
assert f(l0, len(l0), .) == 2
assert f(l0, len(l0), .) == 5
```

1.1.3 Quoi, encore !

Ecrire une fonction récursive de signature identique à celle de `f` qui résout le même problème.

[ ]:
```python
def f(l: list[int], n: int, s: int) -> int:
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

Tests unitaires Vérifier que cette version récursive de `f` vérifie *tous* les tests unitaires de la version itérative.

[ ]:
```python
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

1.2 Exercice 2. Ordre lexicographique et tri associé

Ordre alphabétique. Les caractères alphabétiques en minuscule sont naturellement ordonnés selon l'ordre alphabétique.

- Ainsi, le caractère 'a' est avant le caractère 'b', lui-même avant le caractère 'c' ...
- En python, l'opérateur de comparaison **< appliqué à 2 caractères** vérifie cet ordre.
- On peut ainsi dire que 'a' est strictement inférieur à 'b', 'b' strictement inférieur à 'c', ... et ce selon l'ordre alphabétique.

[ ]:
```python
assert 'a' < 'b'
assert 'b' < 'c'
assert 'a' < 'c'
assert ('c' < 'a') == False
```

Ordre lexicographique. L'*ordre lexicographique* est l'ordre dans lequel **les mots du dictionnaire** sont rangés (triés).

Dans cet exercice, les mots : - sont composés de caractères alphabétiques en minuscule, - et sont de longueur au moins égale à 2.

Remarques.

- La longueur d'un mot est son nombre de caractères.
- Le mot de longueur 0 est le mot vide. Il n'est pas considéré dans cet exercice.
- Un mot de longueur 1 est un caractère et les caractères sont ordonnés selon l'ordre alphabétique (rappel).

Exemple de mots ordonnés selon l'ordre lexicographique.

La liste suivante, utile par la suite, définit 6 mots ordonnés comme dans un dictionnaire. On peut ainsi dire que cette liste de mots **est triée selon l'ordre lexicographique**.

[ ]:
```
mots_test = ['bonjour', 'par', 'part', 'partie', 'parts', 'zebre']
```

Ainsi pour l'ordre lexicographique :

- 'bonjour' est strictement inférieur à 'par',
- 'par' est strictement inférieur à 'part',
- 'bonjour' est strictement inférieur à 'zebre', ...

Mais attention :

- 'par' **n'est pas** strictement inférieur à 'par', ...

Abus de langage.

Par la suite, on ne répétera pas que la comparaison de deux mots s'effectue selon l'ordre lexicographique. On dira par exemple simplement : 'bonjour' est strictement inférieur à 'par'.

Consignes importante.

- Vous pouvez utiliser l'opérateur de comparaison python **< uniquement entre deux caractères**.
  - il vérifie l'ordre alphabétique.
- En revanche, il est **interdit** d'utiliser l'opérateur python **< entre deux chaines de caractères**.
  - cet opérateur "< entre deux chaines de caractères" vérifie l'ordre lexicographique
  - il sera **uniquement** utilisé dans les cellules d'auto-validation.

1.2.1 Ordre lexicographique itératif

On rappelle qu'un mot est une chaine de caractères minuscules de longueur au moins 2.

La fonction `inf_lex_it()` suivante vérifie si le mot `m1` est strictement inférieur à `m2`; `m1` et `m2` étant de longueur respective `n1` et `n2`.

[ ]:
```
def inf_lex_it(m1: str, n1: int, m2: str, n2: int) -> bool:
    '''m1 < m2 : ordre lexicographique, version iterative'''
    pass
```

Tests unitaires avec `mots_test`. Ecrire les 4 tests unitaires de la fonction `inf_lex_it()` donnés plus haut comme exemples de l'ordre lexicographique sur des mots de `mots_test`.

[ ]:
```
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

`inf_lex_it()` Compléter la signature précédente pour définir la fonction `inf_lex_it()`.

[ ]:
```
def inf_lex_it(m1: str, n1: int, m2: str, n2: int) -> bool:
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

Auto-validation

[ ]:
```
assert inf_lex_it("abc", 3, "abd", 3) == ("abc" < "abd")
assert inf_lex_it("bac", 3, "bad", 3) == ("bac" < "bad")
assert inf_lex_it("ab", 2, "abc", 3) == ("ab" < "abc")
assert inf_lex_it("ab", 2, "ab", 2) == (not("ab" < "ab"))
```

Votre validation Utiliser les tests unitaires avec `mots_test` pour valider votre fonction.

[ ]:
```
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

1.2.2 Ordre lexicographique récursif

Ecrire la fonction `inf_lex_rec()`, version récursive de la fonction `inf_lex_it()`.

[ ]:
```
def inf_lex_rec(m1: str, n1: int, m2: str, n2: int) -> bool:
        # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

Validation Ré-écrire les tests unitaires de la version itérative pour cette version récursive et les appliquer.

[ ]:
```
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

1.2.3 Tri de mots selon l'ordre lexicographique

On suppose des mots stockés dans une liste – comme `mots_test` par exemple.

(⋆) **`trier_mots()`** Ecrire `trier_mots()` en reprennant un des algorithmes de tri vus en cours pour trier un telle liste de mots selon l'ordre lexicographique.

[ ]:
```
def trier_mots(l : list[str], n: int) -> list[str]:
    '''retourne la liste l triée selon ordre lexicographique'''
    pass
```

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

[ ]:
```
from random import shuffle

tmp = mots_test[:]  # duplication
shuffle(tmp) # melange en place
print(tmp) # pour vous rassurer

assert trier_mots(tmp, len(tmp)) == mots_test
```

1.3 3. Le jeu OTHELLO ou REVERSI : une version itérative

Description du jeu Othello (aussi connu sous le nom Reversi) est un jeu de société opposant deux joueurs.

Il se joue sur un plateau, appelé *othellier*, de 64 cases (8x8) et avec 64 pions bicolores, noirs d'un côté et blancs de l'autre. En début de partie, quatre pions sont déjà placés au centre de l'othellier comme sur l'affichage ci-dessous (deux noirs en e4 et d5 et deux blancs en d4 et e5). Chaque joueur choisit une couleur : noir ou blanc. Les noirs commencent. Puis chaque joueur pose l'un après l'autre un pion de sa couleur sur l'othellier selon des règles précisées dans le paragraphe suivant. Le jeu s'arrête quand les deux joueurs ne peuvent plus poser de pion.
Le joueur ayant le plus grand nombre de pions de sa couleur sur l'othellier a gagné.

Règles du jeu

- Noir commence toujours la partie.
- A tour de rôle, chaque joueur place un pion qui **doit capturer** des pions adverses.
- Les joueurs jouent à tour de rôle, chacun étant tenu de capturer des pions adverses lors de son mouvement.
- Si un joueur ne peut pas capturer de pion(s) adverse(s), il est forcé de passer son tour.
- Si aucun des deux joueurs ne peut jouer, ou si l'othellier ne comporte plus de case vide, la partie s'arrête.
- Le gagnant en fin de partie est celui qui possède le plus de pions.

La capture de pions : - Elle survient lorsqu'un joueur place un de ses pions à l'extrémité d'un alignement de pions adverses contigus dont l'autre extrémité est déjà occupée par un de ses propres pions. - Les alignements peuvent être selon une colonne, une ligne ou une diagonale. - Si le pion nouvellement placé ferme plusieurs alignements, il capture tous les pions adverses de ces alignements. - La capture se traduit par le retournement des pions capturés (qui prennent alors la couleur du joueur). - Ces retournements n'entraînent pas d'effet de capture en cascade : seul le pion nouvellement posé est pris en compte.

Numérotation de la grille : - les colonnes sont numérotées de gauche à droite par les lettres a à h, - les lignes sont numérotées de haut en bas par les chiffres 1 à 8.

La première figure ci-dessous montre la position de départ où `X` représente un pion noir, `O` un pion blanc et `.` une case vide.

Grille initiale :

```
    ========================
     a  b  c  d  e  f  g  h
    ========================
1 |  .  .  .  .  .  .  .  .
2 |  .  .  .  .  .  .  .  .
3 |  .  .  .  .  .  .  .  .
4 |  .  .  .  O  X  .  .  .
5 |  .  .  .  X  O  .  .  .
6 |  .  .  .  .  .  .  .  .
7 |  .  .  .  .  .  .  .  .
8 |  .  .  .  .  .  .  .  .
```

```
    ========================
    Noirs:  2  Blancs:  2
```

La deuxième figure montre les 4 cases `?` où Noir peut jouer grâce à la capture d'un pion Blanc.

```
    ========================
     a  b  c  d  e  f  g  h
    ========================
1 |  .  .  .  .  .  .  .  .
2 |  .  .  .  .  .  .  .  .
3 |  .  .  .  ?  .  .  .  .
4 |  .  .  ?  O  X  .  .  .
5 |  .  .  .  X  O  ?  .  .
6 |  .  .  .  .  ?  .  .  .
7 |  .  .  .  .  .  .  .  .
8 |  .  .  .  .  .  .  .  .
    ========================
    Noirs:  2  Blancs:  2
```

Noir joue en d3. La figure suivante montre : - le plateau après la capture du pion Blanc d4 (pion retourné devenant ainsi un pion Noir). - les cases `?` où Blanc peut ensuite jouer.

```
    ========================
     a  b  c  d  e  f  g  h
    ========================
1 |  .  .  .  .  .  .  .  .
2 |  .  .  .  .  .  .  .  .
3 |  .  .  ?  X  ?  .  .  .
4 |  .  .  .  X  X  .  .  .
5 |  .  .  ?  X  O  .  .  .
6 |  .  .  .  .  .  .  .  .
7 |  .  .  .  .  .  .  .  .
8 |  .  .  .  .  .  .  .  .
    ========================
    Noirs:  4  Blancs:  1
Joueur 2 :
```

1.3.1 Modélisation du jeu

On modèlisera le joueur au pion noir par l'entier 1 et le joueur au pion blanc pour l'entier 2.

Le plateau de jeu sera modélisé par un tableau en 2 dimensions d'entiers (`list[list[int]]`). Où chaque case pourra prendre les valeurs suivantes :

- 0 pour une case vide,
- 1 pour une case avec un pion noir,
- 2 pour une case avec un pion blanc.

1.3.2 Modélisation de la validation d'une position de jeu pour un joueur

Lorsque un joueur entrera les coordonnées d'une case dans laquelle il souhaite placer un pion, il faudra procéder à la vérification de la case : cette case devra existée, être vide et il devra y avoir au moins un alignement non discontinue de pions du joueur adverse entre cette case et une case du plateau appartenant au même joueur se situant dans une des huits directions (nord, nord-est, est, sud-est, sud, sud-ouest, ouest, nord, ouest).

Les 8 directions peuvent être modélisées par les vecteurs suivants :

- `nord       : [-1,  0]`,
- `nord-est   : [-1, +1]`,
- `est        : [ 0, +1]`,
- `sud-est    : [+1, +1]`,
- `sud        : [+1,  0]`,
- `sud-ouest  : [+1, -1]`,
- `ouest      : [ 0, -1]`,
- `nord-ouest : [-1, -1]`.

Nous pouvons donc construire la liste des vecteurs de directions :

- `[ [-1,  0], [-1, +1], [ 0, +1], [+1, +1], [+1,  0], [+1, -1], [ 0, -1], [-1, -1] ]`

Lorsqu'un joueur souhaite placer un pion aux coordonnées (`l`,`c`), nous utiliserons cette liste afin de parcourir les 8 directions. Pour chaque direction `dv`, nous rechercherons un alignement de pions adverses consécutifs compris entre deux pions du joueur au coordonnées (`l`,`c`) et (`l+k*dv[0]`,`c+k*dv[1]`) avec `k > 1`.

1.3.3 Cases voisines

Ecrire une fonction `couleur_voisin()` qui prend en paramètre une grille `g` de taille `nxn`, `n` un entier, des coordonnées valides (`l`,`c`), un vecteur de direction `dv` et une couleur `couleur`. La fonction vérifie si la case voisine dans la direction `dv` existe et si celle-ci est bien de la couleur `couleur`.

[ ]:
```python
def couleur_voisin(g: list[list[int]], n: int, l: int, c:int, dv: list[int],␣
↪couleur: int) -> bool:
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

1.3.4 Auto-validation

[ ]:
```python
gtest = [
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 2 ,1, 0, 0, 0],
    [0, 0, 0, 1 ,2, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
]
```

```
assert couleur_voisin(gtest, 8, 3, 3, [1,0], 1)==True
assert couleur_voisin(gtest, 8, 3, 3, [1,0], 2)==False
assert couleur_voisin(gtest, 8, 3, 3, [0,1], 1)==True
assert couleur_voisin(gtest, 8, 3, 3, [1,1], 2)==True
```

1.3.5 Affichage simple

Ecrire une fonction simple `afficher_grille(g: list[list[int]])` qui prend en paramètre une grille `g` de taille `n` et l'affiche avec des `.` pour les cases vides, des `X` pour les pions noirs et des `O` pour les pions blancs.

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

[ ]:
```
def afficher_grille(g: list[list[int]], n: int) -> None:
    '''Affiche la grille "g" avec des "." des "X" et  des "O"'''
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

1.3.6 Compter des pions

Un joueur veut placer un pion de couleur `c` sur la case (`lig , col`) **supposée vide**.

La fonction `nb_pion_dir()` :

1. vérifie, dans la direction `dv`, si il existe un alignement de pions adverses consécutifs qui se termine par un pion de couleur `c`,
2. renvoie le nombre de pions adverses consécutifs alignés,
3. ou renvoie -1 si un tel alignement n'existe pas.

Ainsi, le joueur vérifie avec `nb_pion_dir()` si il existe au moins un alignement de pions adverses dans une des 8 directions pour placer un pion de couleur `c` sur la case (`lig , col`).

Écrire le corps de la fonction `nb_pion_dir(g: list[list[int]], n : int, lig: int, col:int, dv: list[int], player: int) -> int`

[ ]:
```
def nb_pion_dir(g: list[list[int]], n: int, lig: int, col:int, dv: list[int],␣
↪couleur: int) -> int:
    '''Fonction qui vérifie si il existe un alignement de pions adverse dans la␣
↪direction dvec à partir de lig, col inclus.
        Retourne le nombre de pions adverses ou -1 s'il n'y a pas d'alignement.
↪'''
```

```
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

1.3.7 Auto-validation

[ ]:
```
gtest = [
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 2 ,1, 0, 0, 0],
    [0, 0, 0, 1 ,2, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
]

d = [1,0]
assert nb_pion_dir(gtest, 8, 2, 3, d, 1)==1
d = [0,1]
assert nb_pion_dir(gtest, 8, 3, 2, d, 1)==1

d = [1,0]
assert nb_pion_dir(gtest, 8, 2, 4, d, 2)==1
d = [-1,0]
assert nb_pion_dir(gtest, 8, 5, 3, d, 2)==1

d = [1,0]
assert nb_pion_dir(gtest, 8, 2, 3, d, 2)==0
d = [0,1]
assert nb_pion_dir(gtest, 8, 3, 2, d, 2)==0

d = [1,0]
assert nb_pion_dir(gtest, 8, 2, 4, d, 1)==0
d = [-1,0]
assert nb_pion_dir(gtest, 8, 5, 3, d, 1)==0

d = [1,0]
assert nb_pion_dir(gtest, 8, 0, 0, d, 1)==-1
d = [-1,0]
assert nb_pion_dir(gtest, 8, 7, 7, d, 1)==-1
```

1.3.8 Valider une position

Ecrire la fonction `pos_valide(g: list[list[int]], lig: int, col:int, couleur: int) -> int:` qui vérifie si les coordonnées (`lig`, `col`) sont valides pour le joueur `couleur` c'est à dire s'il existe un alignement de pions adverses encadré à partir de ces coordonnées dans une des 8 directions possibles.

[ ]:
```
def pos_valide(g: list[list[int]], n: int, lig: int, col: int, couleur: int) ->␣
↪bool:
    '''Fonction qui vérifie s'il existe un alignement consécutif de pions␣
↪adverses au joueur "couleur"
        dans les 8 directions dans la grille g à partir des (lig,col) et␣
↪retourne True si il existe au moins un alignement'''
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

1.3.9 Auto-validation

[ ]:
```
gtest = [
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 2 ,1, 0, 0, 0],
    [0, 0, 0, 1 ,2, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
]

assert pos_valide(gtest, 8, 2, 3, 1)==True
assert pos_valide(gtest, 8, 3, 2, 1)==True

assert pos_valide(gtest, 8, 2, 4, 2)==True
assert pos_valide(gtest, 8, 5, 3, 2)==True

assert pos_valide(gtest, 8, 2, 3, 2)==False
assert pos_valide(gtest, 8, 3, 2, 2)==False

assert pos_valide(gtest, 8, 2, 4, 1)==False
assert pos_valide(gtest, 8, 5, 3, 1)==False

assert pos_valide(gtest, 8, 0, 0, 1)==False
assert pos_valide(gtest, 8, 7, 7, 1)==False
```

Écrire une fonction `nb_pos_valide(g: list[list[int]], n: int, couleur: int) -> int:` qui retourne le nombre de positions valides dans la grille `g` de taille `nxn` pour le joueur `couleur`

[ ]:
```
def nb_pos_valide(g: list[list[int]], n: int, couleur: int) -> int:
    '''Fonction qui compte et retourne le nombre de positions valides dans la␣
↪grille "g" pour le joueur "couleur"'''
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

1.3.10 Auto-validation

```
[ ]: gtest = [
         [0, 0, 0, 0 ,0, 0, 0, 0],
         [0, 0, 0, 0 ,0, 0, 0, 0],
         [0, 0, 0, 0 ,0, 0, 0, 0],
         [0, 0, 0, 2 ,1, 0, 0, 0],
         [0, 0, 0, 1 ,2, 0, 0, 0],
         [0, 0, 0, 0 ,0, 0, 0, 0],
         [0, 0, 0, 0 ,0, 0, 0, 0],
         [0, 0, 0, 0 ,0, 0, 0, 0],
     ]


     assert(nb_pos_valide(gtest, 8, 1)==4)
     assert(nb_pos_valide(gtest, 8, 2)==4)
```

1.3.11 3.3 Afficher une grille un petit peu plus "smart"

Ecrire une fonction `afficher_pos_grille` qui prends en paramètre une grille `g` de type `list[list[int]]`, un numéro de joueur `couleur` de type `int` et affiche la grille `g` sous la forme suivante où les `X` représentent les pions noirs, les `O` les pions blanc et les `?` indiquent les possibilités de jeu du joueur `couleur` comme le montre l'affichage suivant :

```
    ========================
     a  b  c  d  e  f  g  h
    ========================
1 |  .  .  .  .  .  .  .  .
2 |  .  .  .  .  .  .  .  .
3 |  .  .  .  ?  .  .  .  .
4 |  .  .  ?  O  X  .  .  .
5 |  .  .  .  X  O  ?  .  .
6 |  .  .  .  .  ?  .  .  .
7 |  .  .  .  .  .  .  .  .
8 |  .  .  .  .  .  .  .  .
    ========================
    Noirs:  2  Blancs:  2
```

```
[ ]: def afficher_pos_grille(g: list[list[int]], n: int, couleur: int) -> None:
         '''Fonction qui affiche la grille "g" avec les positions valides pour le␣
      ↪joueur "player" '''
         # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
         raise NotImplementedError()
```

1.3.12 Auto-validation

[ ]:
```
# TEST DE LA FONCTION afficher_pos_grille(...)
gtest = [
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 2 ,1, 0, 0, 0],
    [0, 0, 0, 1 ,2, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
]

afficher_pos_grille(gtest, 8, 1)
afficher_pos_grille(gtest, 8, 2)
```

1.3.13 3.4 Placer les pions

Écrire la fonction `capturer(g: list[list[int]], n: int, lig: int, col: int, couleur: int) -> bool:` qui prend en paramètre une grille `g` de taille `nxn`, les coordonnées `lig, col`, le numéro du joueur `couleur`.

On suppose que les coordonnées (`lig,col`) sont valides. La fonction modifie la grille en capturant et retournant tous les jetons du joueur adverse dans les directions "valides" parmis les 8 possibles.

La fonction renvoie le nombre total de pions capturés.

[ ]:
```
def capturer(g: list[list[int]], n: int, lig: int, col: int, couleur: int) ->␣
 ↪int:
    '''Fonction qui joue eventuellement un pion du joueur "couleur" dans la␣
 ↪grille "g" aux coordonnées (lig,col)
        Retourne True si le pion à pu être joué.'''
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

1.3.14 Auto-validation

[ ]:
```
gtest0 = [
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 2 ,1, 0, 0, 0],
    [0, 0, 0, 1 ,2, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
]
gtest1 = [
```

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

assert capturer(gtest0, 8, 0, 0, 1)==False
assert capturer(gtest0, 8, 2, 3, 1)==True

assert capturer(gtest1, 8, 0, 0, 2)==False
assert capturer(gtest1, 8, 2, 3, 2)==False
```

1.3.15 3.5 Programme principal : le jeu et auto validation

Écrire le programme principal qui va permettre de jouer à deux joueurs à tour de rôle à jeu Othello en utilisant les fonctions précédement définies.

[ ]:
```
# PROGRAMME PRINCIPAL

# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

1.4 4. * Bonus *

1.4.1 4.1 *** Compter d'une autre façon

Si vous avez écrit la fonction `nb_pion_dir(...)` de manière itérative réécrivez-la de manière récursive dans le cas contraire réécrivez-la de manière itérative.

[ ]:
```
def nb_pion_dir2(g: list[list[int]], l: int, c:int, dv: list[int], couleur:␣
↪int) -> int:
    '''Fonction qui vérifie si il existe un alignement de pions adverse dans la␣
↪direction dv à partir de lig, col inclus.
        Retourne le nombre de pions adverses ou -1 s'il n'y a pas d'alignement.
↪'''
    # ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
    raise NotImplementedError()
```

1.4.2 Auto-validation

[ ]:
```
gtest = [
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
    [0, 0, 0, 0 ,0, 0, 0, 0],
```

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

d = [1,0]
assert nb_pion_dir2(gtest, 8, 2, 3, d, 1)==1
d = [0,1]
assert nb_pion_dir2(gtest, 8, 3, 2, d, 1)==1

d = [1,0]
assert nb_pion_dir2(gtest, 8, 2, 4, d, 2)==1
d = [-1,0]
assert nb_pion_dir2(gtest, 8, 5, 3, d, 2)==1

d = [1,0]
assert nb_pion_dir2(gtest, 8, 2, 3, d, 2)==0
d = [0,1]
assert nb_pion_dir2(gtest, 8, 3, 2, d, 2)==0

d = [1,0]
assert nb_pion_dir2(gtest, 8, 2, 4, d, 1)==0
d = [-1,0]
assert nb_pion_dir2(gtest, 8, 5, 3, d, 1)==0

d = [1,0]
assert nb_pion_dir2(gtest, 8, 0, 0, d, 1)==-1
d = [-1,0]
assert nb_pion_dir2(gtest, 8, 7, 7, d, 1)==-1
```

[ ]: