Amérique du sud - septembre 2023 - sujet 2

Exercice 1 (File, pile et POO - 4 points)
On s'intéresse à l'évaluation d'expressions mathématiques, comportant uniquement des additions, des multiplications et des nombres entiers. On utilisera pour cela les structures de pile et de file dont les interfaces sont données ci-dessous :

- Interface de la classe `Pile` :
  - `Pile()` : crée une pile vide
  - `empile(el)` : empile l'élément `el` au sommet de la pile
  - `depile()` : supprime et renvoie l'élément au sommet de la pile ; déclenche une erreur si la pile est vide
  - `est_vide()` : renvoie `True` si la pile est vide, `False` sinon

- Interface de la classe `File` :
  - `File()` : crée une file vide
  - `enfile(el)` : ajoute l'élément `el` à la queue de la file
  - `defile()` : supprime et renvoie l'élément en tête de la file ; déclenche une erreur si la file est vide
  - `est_vide()` : renvoie `True` si la file est vide, `False` sinon

Afin de tenir compte des priorités des opérations, on représente ces expressions par des arbres binaires. Ainsi, l'expression $1+5\times(3+9)$ est représenté par l'arbre suivant :

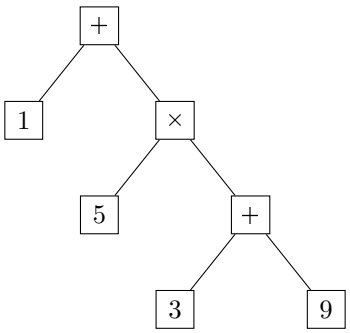


FIGURE 1

1. Donner l'expression représentée par l'arbre ci-dessous :

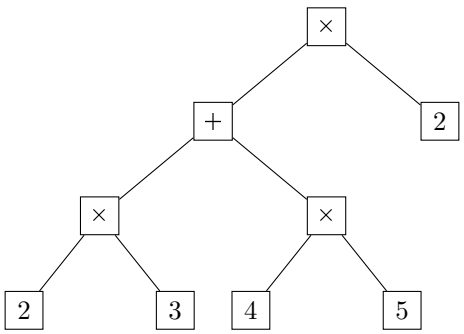


FIGURE 2

2. On décide d'implémenter en Python un arbre binaire à l'aide de la classe `Noeud` ci-dessous.

```
class Noeud:
    def __init__(self, etiquette, gauche, droit):
        self.etiq = etiquette
        self.sag = gauche
        self.sad = droit
```

Un sous-arbre vide sera représenté par `None`.
Dessiner l'arbre `expression` qui est défini par le code suivant :

```
feuille2 = Noeud('2', None, None)
feuille3 = Noeud('3', None, None)
feuille4 = Noeud('4', None, None)
feuille5 = Noeud('5', None, None)
feuille6 = Noeud('6', None, None)
noeud0 = Noeud('*', feuille2, feuille3)
noeud1 = Noeud('+', feuille5, feuille6)
noeud2 = Noeud('+', noeud0, feuille4)
expression = Noeud('*', noeud2, noeud1)
```

3. Le parcours suffixe (aussi appelé postfixe) d'un arbre représentant une expression mathématique permet d'en obtenir une représentation appelée notation polonaise inversée.
   (a) Donner la liste des étiquettes de l'arbre de la question 1 (Figure 2) dans l'ordre du parcours suffixe de cet arbre.
   (b) On donne ci-après trois propositions de fonctions récursives dont le premier paramètre est un arbre représentant une expression mathématique et le second est une file initialement vide. Laquelle de ces fonctions renvoie la file contenant les étiquettes de l'arbre dans l'ordre du parcours suffixe ?

Proposition 1 :

```
def suffixe(arbre, file):
    if arbre != None:
        file.enfile(arbre.etiq)
        parcours_g = suffixe(arbre.sag, file)
        parcours_d = suffixe(arbre.sad, file)
    return file
```

Proposition 2 :

```
def suffixe(arbre, file):
    if arbre != None:
        parcours_g = suffixe(arbre.sag, file)
        file.enfile(arbre.etiq)
        parcours_d = suffixe(arbre.sad, file)
    return file
```

Proposition 3 :

```
def suffixe(arbre, file):
    if arbre != None:
        parcours_g = suffixe(arbre.sag, file)
        parcours_d = suffixe(arbre.sad, file)
        file.enfile(arbre.etiq)
    return file
```

4. L'évaluation d'une expression mathématique consiste à effectuer les différentes opérations pour obtenir le résultat du calcul correspondant. On donne ci-dessous un algorithme permettant d'évaluer une expression donnée sous la forme d'un arbre :
   - ⋆ On effectue un parcours suffixe de cet arbre pour obtenir une file contenant ses étiquettes dans l'ordre de la notation polonaise inversée.
   - ⋆ Pour chaque élément défilé :
     - si c'est un nombre, on l'empile ;
     - si c'est un opérateur (`'+'` ou `'*'`), on dépile les deux éléments `d` et `g` du sommet de la pile, et on empile le résultat de l'opération appliquée à `g` et `d`.
   - ⋆ Lorsque la file est vide, la pile contient un seul élément : le résultat de l'évaluation de l'expression.

   Par exemple, lors de l'évaluation de l'expression en notation polonaise inversée `3 10 + 5 ×`, voici les différentes états de la pile, suite au défilement d'un élément :

Elément défilé	3	10	+	5	×
Pile	3	10 3	13	5 13	65

   Le fonction `evalue` ci-dessous implémente cet algorithme. Elle renvoie le résultat de l'évaluation d'une expression mathématique représentée par un arbre binaire `arbre` passé en paramètre.
   Sur votre copie, recopier et compléter cette fonction.

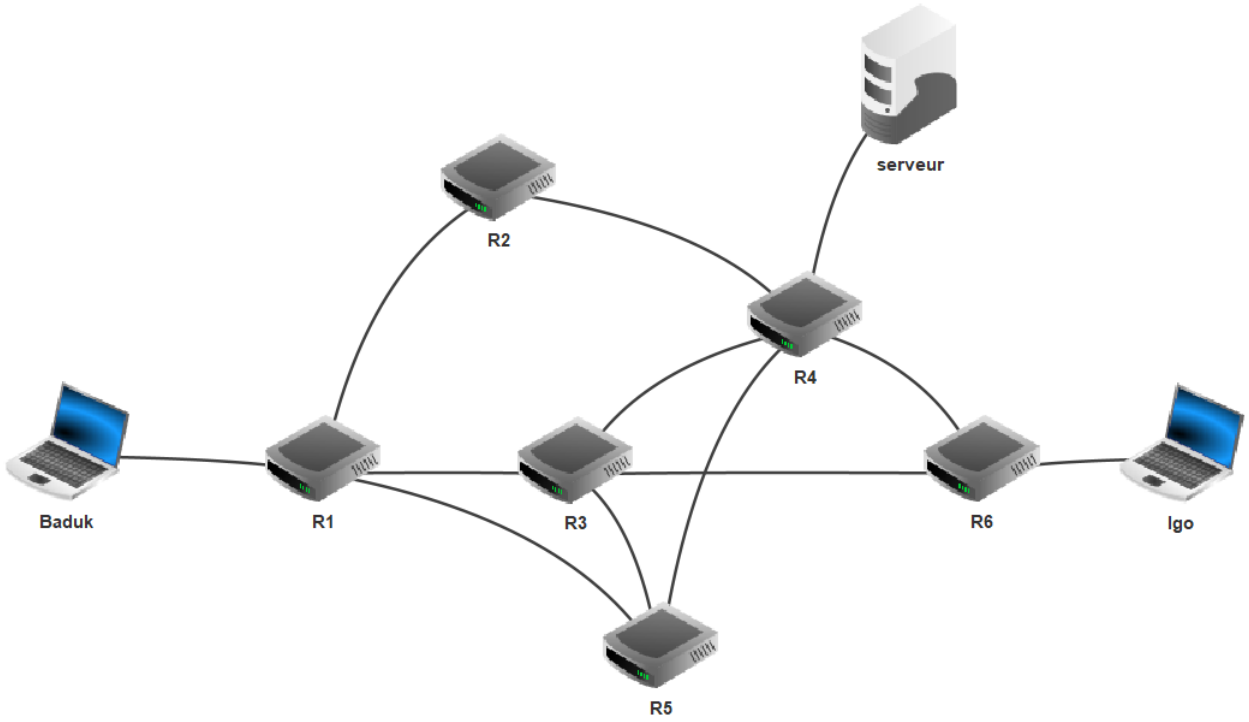
```
def evalue(arbre):
    f = suffixe(arbre, File())
    p = Pile()
    while ... :
        elt = f.defile()
        if elt == '+' or elt == '*':
            ... # plusieurs lignes
        else:
            ...
    return ...
```

   On pourra utiliser la fonction `op` ci-dessous :

```
def op(symbole, x, y):
    if symbole == '+':
        return int(x) + int(y)
    else:
        return int(x) * int(y)
```

Exercice 2 (Routage et bases de données - 4 points)
Le jeu de go est un jeu de société originaire de Chine. Il oppose deux adversaires qui placent à tour de rôle des pierres, respectivement noires et blanches, sur un plateau.

Partie A : réseau.
Baduk et Igo jouent une partie de go grâce à une application en ligne hébergée sur un serveur. La portion de réseau au voisinage des ordinateurs de Baduk, d'Igo et du serveur de jeu est représentée ci-dessous.



1. Les tables de routage des routeurs sont les suivantes. La métrique permettant ici de décider du meilleur chemin vers un routeur distant est le nombre de sauts.

R1		
Destination	**Passerelle**	**Métrique**
R2, R3, R5	–	1
R4	R2	2
R6	R3	2

R2		
Destination	**Passerelle**	**Métrique**
R1, R4	–	1
R3, R5, R6	R4	2

R3		
Destination	**Passerelle**	**Métrique**
R1, R4, R5, R6	–	1
R2	R4	2

R4		
Destination	**Passerelle**	**Métrique**
R2, R3, R5, R6	–	1
R1	R5	2

R5		
Destination	**Passerelle**	**Métrique**
R1, R3, R4	–	1
R2	R1	2
R6	R4	2

R6		
Destination	**Passerelle**	**Métrique**
R3, R4	–	1
R1, R5	R3	2
R2	R4	2

Igo effectue un coup grâce à son application, puis Baduk joue à son tour. Les données partent de la machine d'Igo vers le serveur, puis sont transmises à la machine de Baduk. Les données du coup de Baduk sont envoyées de sa machine vers le serveur, puis transmises à la machine d'Igo.
En utilisant les tables de routage fournies ci-dessus, indiquer un chemin emprunté par les données lors de ces échanges.

2. Recopier la table de routage du routeur R3 avec une mise à jour possible suite à la rupture de la liaison entre les routeurs R3 et R4.

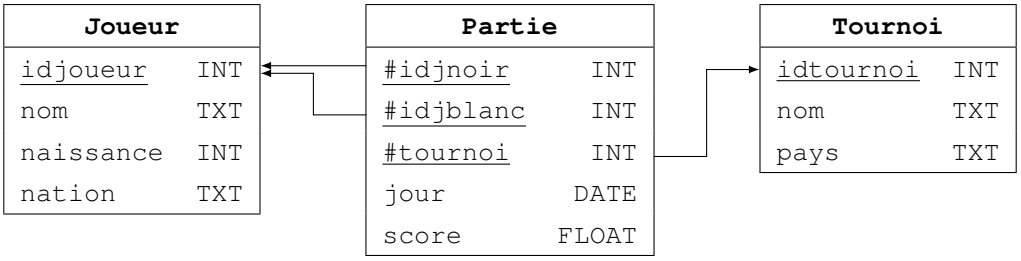
Partie B : base de données.

Dans cette partie, on pourra utiliser les mots clés du langage SQL suivants : `SELECT`, `INSERT INTO`, `DISTINCT`, `WHERE`, `UPDATE`, `JOIN`, `COUNT`, `MIN`, `MAX`, `ORDER BY`.

La fonction d'agrégation `COUNT(*)` renvoie le nombre d'enregistrements de la requête. Les fonctions d'agrégation `MIN(propriété)` et `MAX(propriété)` renvoient respectivement la plus petite et la plus grande valeur de l'attribut `propriété` pour les enregistrements de la requête. La commande `ORDER BY propriété` permet de trier les résultats de la requête selon l'attribut `propriété`.

Le responsable de la fédération internationale de jeu de go enregistre dans une base de données les résultats de parties historiques.

Il définit pour cela des relations `Joueur`, `Partie` et `Tournoi` qui suivent le schéma relationnel suivant (les clés primaires sont soulignées et les clés étrangères sont précédées du caractère #).



Les clés primaires sont soulignées et les clés étrangères sont précédées du caractère #. Ainsi, l'attribut `idjnoir` de la relation `Partie` est une clé étrangère qui fait référence à l'attribut `idjoueur` de la relation `Joueur`.
`idjnoir` et `idjblanc` identifient les joueurs ayant respectivement les pierres noires et les pierres blanches.

3. On suppose que ce schéma relationnel a été implémenté dans un système de gestion de base de données. La base de données est vide et on souhaite enregistrer les résultats d'un premier tournoi à l'aide des commandes SQL suivantes :

```
1  INSERT INTO Joueur(idjoueur, nom, naissance, nation)
2  VALUES (1, 'Dosaku', 1645, 'Japon'),
3         (2, 'Genan Inseki', 1798, 'Japon'),
4         (3, 'Shusaku', 1829, 'Japon');
5
6  INSERT INTO Partie(idjnoir, idjblanc, tournoi, jour, score)
7  VALUES (2, 3, 1, '1846-09-12', -2);
8
9  INSERT INTO Tournoi(idtournoi, nom, pays)
10 VALUES (1, 'Osaka', 'Japon');
```

   (a) Quelle est la nature de l'erreur produite par l'exécution de cette succession de commandes ? Justifier.

   (b) Comment corriger la succession des commandes SQL données pour que le traitement s'effectue sans erreur ?

On donne ci-dessous un extrait des enregistrements contenus dans la base de données.

Joueur			
idjoueur	nom	naissance	nation
1	Dosaku	1645	Japon
2	Genan Inseki	1798	Japon
3	Shusaku	1829	Japon
4	Kitani Minuro	1909	Japon
5	Go Seigen	1914	Chine
6	Sakata Bio	1920	Japon
7	Rin Kaiho	1942	Taiwan
8	Cho Chikun	1953	Corée
9	Rui Naiwei	1963	Chine
10	Lee Changho	1975	Corée

Partie				
idjnoir	idjblanc	tournoi	jour	score
2	3	1	1846-09-12	-2
4	5	0	1933-11-12	0
5	4	2	1939-09-28	2
5	6	0	1953-11-19	999
5	4	3	1961-06-28	999
6	5	3	1962-08-05	0
7	6	3	1967-08-09	2
7	8	4	1983-05-16	-999
10	8	6	1993-04-24	0.5
9	10	5	2000-01-04	999

Tournoi		
idtournoi	nom	pays
0	Inconnu	Autre
1	Osaka	Japon
2	Kamakura games	Japon
3	Meijin	Japon
4	Honinbo	Japon
5	Guksu	Corée
6	Ton Yang Cup	Corée

4. On considère la requête SQL suivante :

```
SELECT COUNT(*) FROM Partie
WHERE idjnoir = 4 OR idjblanc = 4;
```

   (a) Quel serait l'affichage produit par cette requête, appliquée aux seuls extraits des enregistrements donnés dans les tableaux précédents ?

   (b) Expliquer avec une phrase ce que renvoie cette requête.

5. Proposer une requête qui renvoie, dans l'ordre alphabétique, les noms des tournois ayant eu lieu au Japon.

6. Proposer une requête qui renvoie le nom des joueurs qui ont joué avec les pierres noires le 15 mars 2016.

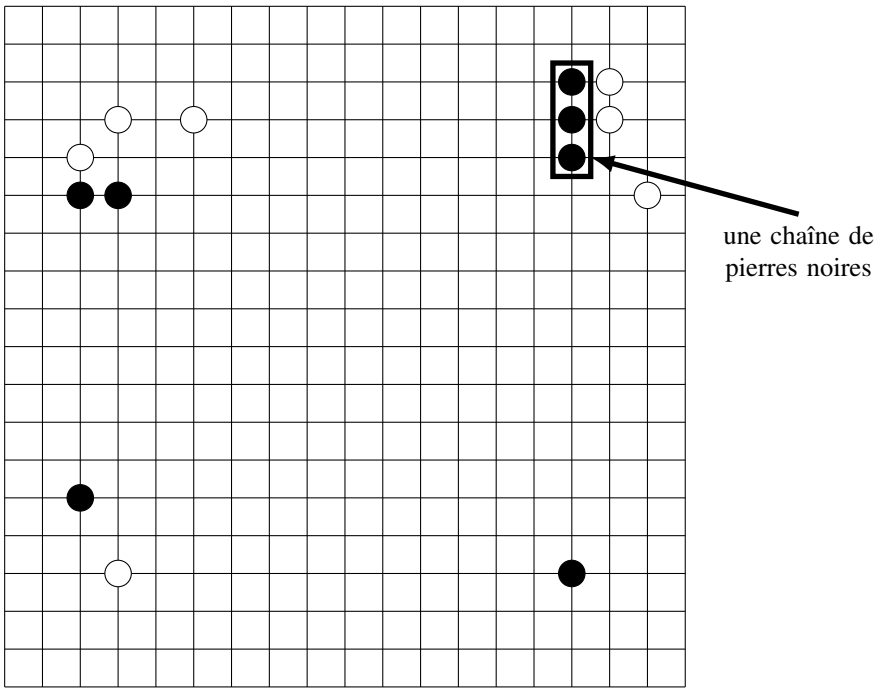
7. Expliquer avec une phrase ce que renvoie la requête suivante :

```
SELECT DISTINCT nom
FROM Joueur
JOIN Partie
ON (Joueur.idjoueur = Partie.idjnoir OR Joueur.idjoueur = Partie.idjblanc)
WHERE Partie.tournoi = 3;
```

Exercice 3 (Tableaux, programmation, récursivité - 4 points)

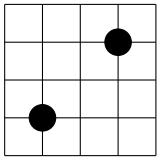
Au jeu de go, les deux adversaires placent à tour de rôle des pierres, respectivement noires et blanches, sur les intersections d'un plateau quadrillé appelé goban. Une partie se joue généralement sur un goban 19×19 intersections, mais des gobans de 13×13 et 9×9 peuvent être utilisés pour s'entraîner.

On a reproduit ci-dessous un début de partie sur un goban de 19×19.

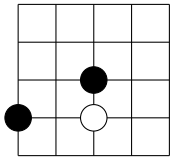


On s'intéresse dans cet exercice à l'une des règles du jeu de go, dite d'encerclement, pour laquelle on doit compter ce qui s'appelle les libertés d'une pierre ou d'une chaîne de pierres.

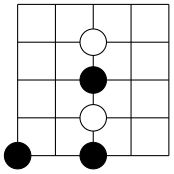
Les libertés d'une pierre sont les intersections libres autour d'elle selon les quatre directions cardinales (nord, sud, est et ouest). Dans chacun des exemples suivants, on donne le nombre de libertés de chacune des pierres noires.



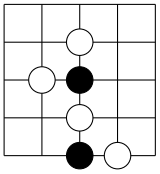
4 libertés



3 libertés

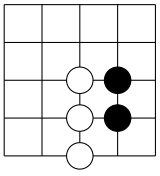


2 libertés

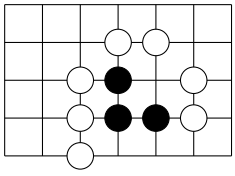


1 liberté

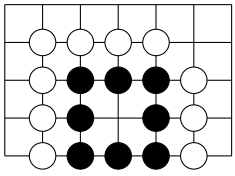
On appelle chaîne de pierres une suite de pierres liées entre elles selon les directions cardinales. Le nombre de libertés d'une chaîne de pierres s'obtient alors en comptant, sans répétition, les libertés de chacune des pierres qui la composent. Les exemples ci-dessous mettent en lumière cette règle sur une chaîne de pierres.



la chaîne de pierres noires a 4 libertés



la chaîne de pierres noires a 3 libertés



la chaîne de pierres noires a 1 liberté

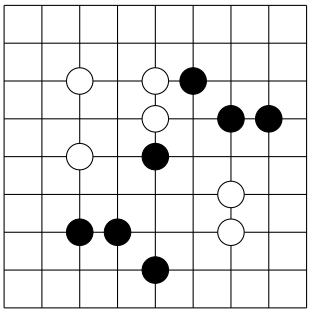
On se propose de représenter un état du jeu sur un goban $n \times n$ (où n peut valoir uniquement 9, 13 ou 19) par une liste de n listes, toutes de longueur n. Chaque intersection sera définie par un numéro de ligne `i` et un numéro de colonne `j`.

L'intersection en haut à gauche (nord-ouest) est repérée par les indices `i = 0` et `j = 0`, tandis que celle en bas à gauche (sud-ouest) est repérée par les indices `i = n - 1` et `j = 0`.

Pour chaque intersection :

- la présence d'une pierre noire est codée par le nombre `1` ;
- la présence d'une pierre blanche est codée par le nombre `-1` ;
- l'absence de pierre est codée par le nombre `0`.

Par exemple, voici un goban de taille 9×9 et sa représentation par la liste `goban` :



```
goban = [[0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0,-1, 0,-1, 1, 0, 0, 0],
         [0, 0, 0, 0,-1, 0, 1, 1, 0],
         [0, 0,-1, 0, 1, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0,-1, 0, 0],
         [0, 0, 1, 1, 0, 0,-1, 0, 0],
         [0, 0, 0, 0, 1, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

1. (a) La valeur `goban[2][5]` indique la présence d'une pierre. Donner sa couleur sans justifier.

   (b) Sur votre copie, recopier et compléter la fonction ci-dessous afin qu'elle renvoie la représentation d'un goban de taille $n \times n$ vide.

```
def creer_goban_vide(n):
    assert n == 9 or n == 13 or n == 19, 'valeur de n non permise'
    ...
```

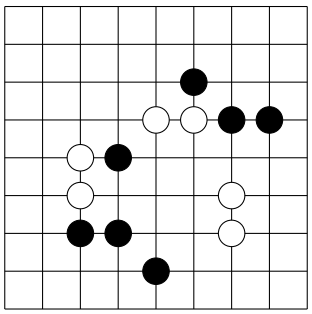
   (c) Que se passe-t-il lors de l'appel `creer_goban_vide(11)` ? Expliquer.

Dans la suite de l'exercice, on aura besoin de la fonction `est_valide` définie par le code suivant :

```
def est_valide(i, j, n):
    return 0 <= i < n and 0 <= j < n
```

Cette fonction renvoie `True` si les indices `i` et `j` passés en paramètres représentent une position valide sur un goban de taille $n \times n$, `False` sinon. Elle pourra être utilisée directement dans la suite de cet exercice.

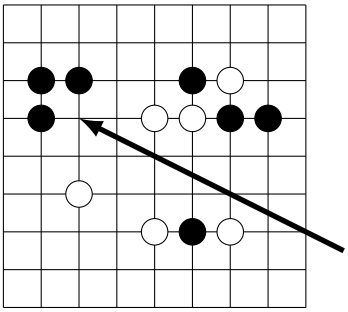
2. La fonction `libertes_pierre` prend pour paramètre une liste `go` représentant un goban et des indices `pi` et `pj` repérant l'intersection où se situe une pierre. Elle renvoie les positions des intersections libres autour de cette pierre sous la forme d'une liste de tuples où chaque tuple est l'une de ces positions.
   Par exemple, si `goban` représente le goban ci-dessous, l'appel `libertes_pierre(goban, 4, 2)` renvoie une liste contenant, dans un ordre quelconque, les tuples `(4, 1)` et `(3, 2)`.



   Sur votre copie, recopier et compléter le code ci-dessous.

```
def libertes_pierre(go, pi, pj):
    libertes = []
    n = len(go)

    ... # plusieurs lignes

    return libertes
```

3. Une chaîne de pierres sera représentée par une liste de tuples qui repèrent leur positions. Afin de calculer le nombre de libertés d'une chaîne de pierres, on examine les libertés, sans répétition, de chacune des pierres qui la composent. On utilise pour cela une liste de liste `marquage` qui contient initialement des booléens tous égaux à `False`. Cette liste `marquage`, qui a les mêmes dimensions que le goban, permet de conserver une trace des libertés qui ne doivent pas être comptées plusieurs fois.
Pour illustrer le rôle de la liste `marquage`, on considère, sur le goban ci-dessous, la chaîne formée des pierres dont les positions sont données par les tuples `(3, 1)`, `(2, 1)` et `(2, 2)`.



En supposant que le premier tuple examiné est `(3, 1)`, la liste `marquage` contiendra alors les valeurs ci-dessous.

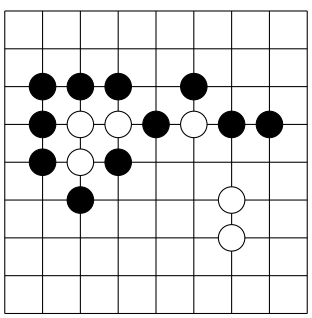
```
[[False, False, False, False, False, False, False, False, False],
 [False, False, False, False, False, False, False, False, False],
 [False, False, False, False, False, False, False, False, False],
 [True , False, True , False, False, False, False, False, False],
 [False, True , False, False, False, False, False, False, False],
 [False, False, False, False, False, False, False, False, False],
 [False, False, False, False, False, False, False, False, False],
 [False, False, False, False, False, False, False, False, False],
 [False, False, False, False, False, False, False, False, False]]
```

Ainsi, la liberté correspondant au tuple `(3, 2)` ne sera pas comptée une nouvelle fois lors de l'étude du tuple `(2, 2)` car `marquage[3][2]` vaut déjà `True`.

La fonction `nb_liberte_chaine` ci-dessous prend en paramètres une liste `go` représentant un goban et une liste `chaine` représentant une chaîne de pierres. Elle renvoie le nombre de libertés de cette chaîne.
Sur votre copie, écrire la séquence d'instructions qui sera exécutée dans la boucle `for i, j in libertes_pierre(go, pi, pj)`.

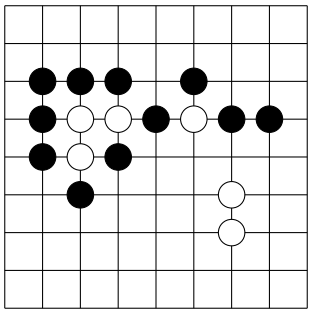
```
def nb_liberte_chaine(go, chaine):
    n = len(go)
    marquage = [[False for j in range(n)] for i in range(n)]
    nb_libertes = 0
    for pos in chaine:
        pi = pos[0]
        pj = pos[1]
        for i, j in libertes_pierre(go, pi, pj):

            ... # plusieurs lignes

    return nb_libertes
```

4. Lorsqu'une chaîne ne possède aucune liberté, on dit que les pierres qui la constituent sont prisonnières, et celles-ci sont alors retirées du goban. Dans l'exemple ci-dessous, la chaîne formée des trois pierres situées aux intersections repérées par les tuples `(4, 2)`, `(3, 2)` et `(3, 3)` n'ont plus de libertés : elles sont donc prisonnières et sont retirées du goban.



On souhaite écrire une fonction qui, si le nombre de libertés d'une chaîne est nul, renvoie le nombre de pierres prisonnières et supprime ces pierres du goban. Ecrire une telle fonction `supprime_prisonniers` qui prend en paramètres une liste `go` représentant un goban et une liste `chaine` représentant une chaîne de pierres.

5. On souhaite maintenant écrire une fonction `cherche_chaine` qui construit, étant donnée la position `(pi, pj)` d'une pierre, la chaîne de pierres qui contient cette pierre. Pour cela, on examine récursivement les pierres voisines de même couleur et on ajoute leur position à une liste `chaine` initialement vide.
Sur votre copie, recopier et compléter les lignes 6 et 7 de cette fonction.

```
1 def cherche_chaine(go, pi, pj, chaine):
2     n = len(go)
3     chaine.append((pi, pj))
4     couleur = go[pi][pj]
5     for i, j in [(pi+1, pj), (pi-1, pj), (pi, pj+1), (pi, pj-1)]:
6         if est_valide(i, j, n) and ... and ... :
7             cherche_chaine(...)
8     return chaine
```

Par exemple, pour le goban ci-dessous, l'appel `cherche_chaine(goban, 3, 1, [])` renvoie la liste `[(3, 1), (4, 1), (2, 1), (2, 2), (2, 3)]`.