

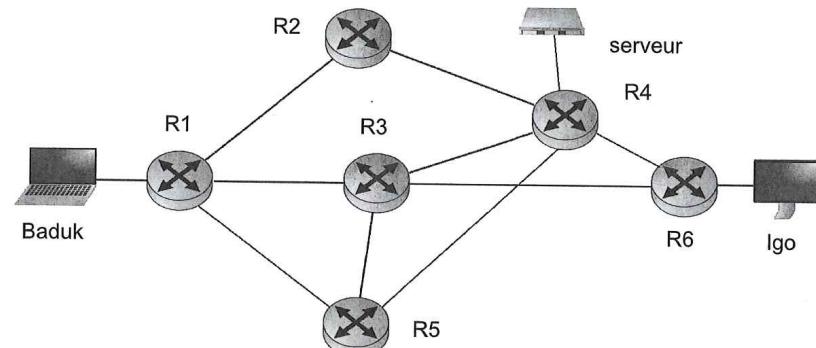
Exercice 2 (4 points)

Les deux parties de cet exercice sont indépendantes.

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

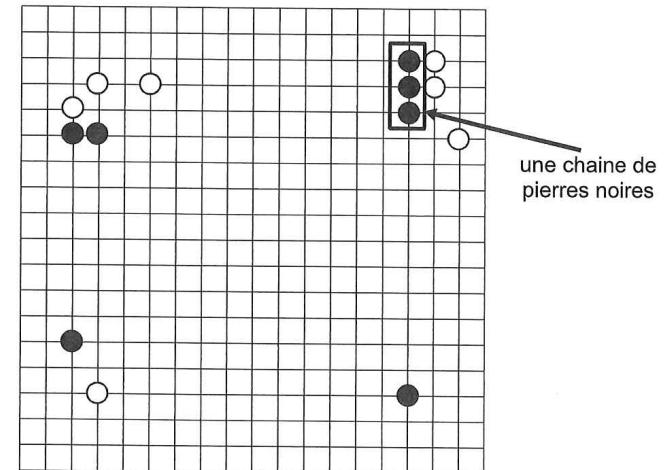
Igo effectue un coup grâce à son application puis Baduk joue à son tour. Les données partent de la machine d'Igo vers le serveur puis sont transmises à la machine de Baduk. Les données du coup de Baduk sont envoyées de sa machine vers le serveur puis transmises à la machine d'Igo.

En utilisant les tables de routage fournies, indiquer un chemin emprunté par les données lors de ces échanges.

Exercice 3 (4 points)

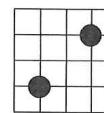
Au jeu de go, les deux adversaires placent à tour de rôle des pierres, respectivement noires et blanches, sur les intersections d'un plateau quadrillé appelé goban. Une partie se joue généralement sur un goban de 19×19 intersections, mais des gobans de 13×13 et 9×9 peuvent être utilisés pour s'entraîner.

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

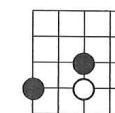


On s'intéresse dans cet exercice à l'une des règles du jeu de go, dite d'encerclement, pour laquelle on doit compter ce qui s'appelle les libertés d'une pierre ou d'une chaîne de pierres.

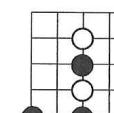
Les libertés d'une pierre sont les intersections libres autour d'elle selon les quatre directions cardinales (nord, sud, est et ouest). Dans chacun des exemples suivants, on donne le nombre de libertés de chacune des pierres noires.



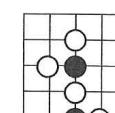
4 libertés



3 libertés



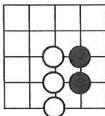
2 libertés



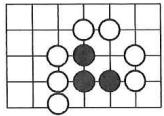
1 liberté

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.

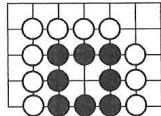
Les exemples ci-dessous mettent en lumière cette règle sur une chaîne de pierres.



La chaîne de pierres noires a 4 libertés.



La chaîne de pierres noires a 3 libertés.



La chaîne de pierres noires a une seule liberté.

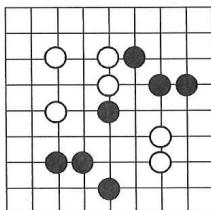
On se propose de représenter un état du jeu sur un goban $n \times n$ (où n peut valoir uniquement 9, 13 ou 19) par une liste de n listes toutes de longueur n . Chaque intersection sera définie par un numéro de ligne i et un numéro de colonne j .

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,
- celle d'une pierre blanche par le nombre -1,
- l'absence de pierre par le nombre 0.

Par exemple, voici un goban de 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.

- La valeur de `goban[2][5]` indique la présence d'une pierre : donner sa couleur sans justifier.
- Sur votre copie, recopier et compléter la fonction ci-dessous afin qu'elle renvoie la représentation d'un goban de $n \times n$ vide.

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

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

Pour la suite de l'exercice, on aura besoin de la fonction `est_valide(i, j, n)` définie par le code suivant.

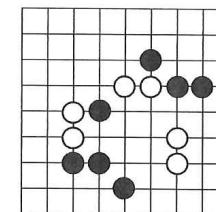
```
def est_valide(i, j, n):
    return 0<=i<n and 0<=j<n
```

Cette fonction renvoie la valeur True si les indices i et j passés en paramètres représentent une position valide sur un goban $n \times n$, et False sinon.

Cette fonction pourra être utilisée dans la suite de cet exercice.

- La fonction `libertes_pierre(go, pi, pj)` a pour paramètres une liste `go` représentant un goban et des indices `pi` et `pj` qui repèrent l'intersection où se situe une pierre. Cette fonction 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 qui contient, dans un ordre quelconque, les tuples `(4, 1)` et `(3, 2)`.



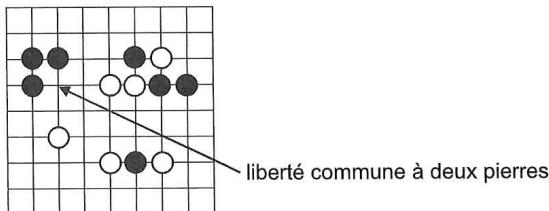
Sur votre copie, recopier et compléter le code ci-dessous.

```
def libertes_pierre(go, pi, pj):
    libertes = []
    n = len(go)
    ... # plusieurs lignes
    return libertes
```

- Une chaîne de pierres sera représentée par une liste de tuples qui repèrent leurs 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 listes `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 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],  
 [True , False , False, False, False, False, False],  
 [False, True , 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.

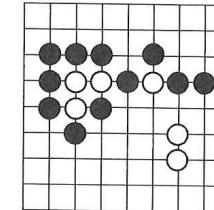
La fonction nb_liberte_chaine(go, chaine) prend en paramètres une liste go représentant un goban et une liste chaine représentant une chaîne de pierres, et renvoie le nombre de libertés de la chaîne.

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 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.

Écrire une telle fonction supprime_prisonniers(go, chaine). Elle a pour 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 leurs positions à une liste chaine initialement vide.

Sur votre copie, recopier et compléter les lignes 48 et 49 de cette fonction.

```
42 def cherche_chaine(go, pi, pj, chaine):  
43     n = len(go)  
44     chaine.append((pi, pj))  
45     couleur = go[pi][pj]  
46     for i, j in [(pi+1, pj), (pi-1, pj),  
47                   (pi, pj+1), (pi, pj-1)]:  
48         if est_valide(i, j, n) and ... and ... :  
49             cherche_chaine( ... )  
50     return chaine
```

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

