A2026 – INFO COMMUNE

ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA,
TÉLÉCOM PARIS, MINES PARIS - PSL,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2026

ÉPREUVE D'INFORMATIQUE COMMUNE

Durée de l'épreuve : 2 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

Les candidats sont priés de mentionner de façon apparente sur la première page de la copie :

INFORMATIQUE COMMUNE

Cette épreuve est commune aux candidats des filières MP, PC et PSI.

L'énoncé de cette épreuve comporte 8 pages de texte.

Le travail doit être reporté sur le cahier de réponses de 8 pages distribué avec le sujet. Un seul cahier de réponses est fourni au candidat, dont toutes les feuilles seront obligatoirement rendues à la fin de l'épreuve. Le renouvellement de ce document en cours d'épreuve est interdit.

Pour valider ce cahier réponses, chaque candidat doit obligatoirement y inscrire à l'encre, à l'intérieur du rectangle d'anonymat situé en haut de chaque copie, sa date de naissance, son nom, son prénom, son numéro d'inscription et sa signature.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Jeu Quixo à deux joueurs.

On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction. La signature des fonctions ne doit pas être rappelée sur la copie. Sauf mention contraire, il n'est pas autorisé d'utiliser des fonctions internes à Python de complexité linéaire sur les listes, dictionnaires et chaînes de caractères excepté le tranchage ou extraction.

1 Présentation

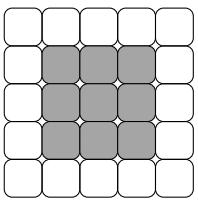
Quixo est un jeu de société qui se joue sur un plateau carré de 5x5 cases. Le jeu comporte 25 cubes identiques possédant 4 faces neutres (blanches), une face marquée d'une croix X et une face marquée d'un rond O. Initialement le plateau de jeu est préparé en mettant tous les cubes sur une face neutre.

Le joueur 1 prend le symbole X et le joueur 2 le symbole O. Les joueurs jouent à tour de rôle. A chaque tour :

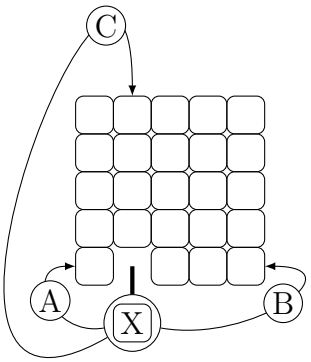
— le joueur choisit l'un de ses cubes ou un cube neutre ; le cube choisi est obligatoirement situé sur les bords du plateau (cases blanches sur la figure 1(a)),

— le cube est ensuite replacé, en le passant à la marque du joueur s'il était neutre, pour pousser les autres cubes jusqu'à boucher la case libérée précédemment (figure 1(b)). Il est interdit de remettre le cube à sa place d'origine.

Le premier joueur à aligner, selon une ligne, une colonne ou une diagonale, cinq de ses symboles gagne la partie.



(a) Cubes utilisables sur le bord



(b) Mouvements possibles à partir du cube choisi

FIGURE 1 – Illustration du plateau de jeu

Dans tout le sujet, pour simplifier, on appelle pion X un cube orienté selon la marque du joueur 1, pion O un cube orienté selon la marque du joueur 2, et pion neutre un cube sur une face neutre.

2 Jeu à deux joueurs humains

Le plateau de jeu est représenté par une liste de listes de dimension 5×5 contenant les valeurs : 0 pour une face neutre, 1 pour la face X du joueur 1 et 2 pour la face O du joueur 2.

Dans le sujet, étant donné que la taille du plateau est fixe, il est possible d'utiliser directement la valeur 5 plutôt que `len(plateau)`.

❏ **Q1** – Écrire une fonction `initialisation() -> [[int]]` qui initialise le plateau de jeu 5×5 avec uniquement des cases neutres.

Le choix, naïf, retenu pour stocker les différents plateaux de jeu est gourmand en mémoire car il nécessite 25 entiers (chacun étant codé sur 2 octets soit 16 bits). Pour optimiser le stockage, on pourrait associer un nombre entier à chaque configuration de plateau.

❏ **Q2** – Sans tenir compte d'éventuelles symétries ou de configurations inaccessibles, déterminer une borne supérieure du nombre de configurations possibles du plateau de jeu. À l'aide d'un logarithme, en déduire une expression du nombre de bits nécessaires pour représenter ces configurations par des entiers.

Pour la suite des questions, on ne s'occupe pas du stockage et on considère que l'on manipule une liste de listes d'entiers.

Soit le plateau de jeu de la figure 2. La case d'indice (0,0) est située en haut à gauche.

❏ **Q3** – Donner les couples d'indices (ligne, colonne) valides ordonnés par indice de ligne croissant correspondant aux pions que peut choisir le joueur 1.

❏ **Q4** – Le joueur 1 choisit le pion de coordonnées (1,4). Donner les cases où il peut repositionner son pion pour finir son tour de jeu.

	0	1	2	3	4
0	O	O	O	X	
1	O				X
2	X	O	X	O	O
3	O		X		X
4	O	X	O	X	O

FIGURE 2 – Situation de jeu - Le joueur 1 doit jouer

Nous allons programmer les fonctions élémentaires correspondant à chaque situation d'un tour de jeu. Tout d'abord, le joueur dont c'est le tour, choisit les coordonnées du pion qu'il souhaite déplacer. Il faut vérifier que le pion est sur le bord et que c'est un pion neutre ou à sa marque.

❏ **Q5** – Écrire une fonction `case_bord(i:int, j:int) -> bool` qui prend en arguments les coordonnées i et j d'une case et qui renvoie `True` si la case appartient bien au bord du plateau et `False` sinon. Il n'est pas autorisé d'énumérer explicitement les coordonnées de toutes les cases du bord.

❏ **Q6** – Écrire une fonction `case_choix_valide(jeu:[[int]], i:int, j:int, joueur:int) -> bool` qui prend en arguments le plateau de jeu, les coordonnées i et j de la case que le joueur a choisie ainsi qu'un entier `joueur` qui correspond au numéro du joueur (1 ou 2). Cette fonction renvoie `True` si la case est valide et `False` sinon. Vous réutiliserez obligatoirement la fonction `case_bord` précédente.

Le joueur, dont c'est le tour, donne les coordonnées où il souhaite repositionner le pion.

❏ **Q7** – Écrire une fonction
`case_deplacement_valide(i_d:int, j_d:int, i_n:int, j_n:int) -> bool` qui prend en arguments les coordonnées de départ i_d et j_d du pion, supposées valides, et les nouvelles coordonnées i_n et j_n et qui renvoie `True` si la nouvelle case choisie est correcte et `False` sinon.

Il faut ensuite modifier le plateau de jeu en faisant glisser les pions vers le bas, vers le haut, vers la gauche ou vers la droite afin de boucher la case de départ du pion.

On donne le code incomplet qui réalise cette procédure sur le document réponse.

❏ **Q8** – Déterminer quel est le mouvement global des pièces pour les quatre cas définis dans la fonction au niveau des commentaires "mouvement 1", "mouvement 2", "mouvement 3" et "mouvement 4". Répondre dans la zone blanche à côté du commentaire.
Montrer que la procédure se termine en n'étudiant que le cas du mouvement 1. On précisera le variant de boucle retenu.

❏ **Q9** – Compléter les lignes vides de la procédure précédente (mouvement 2).

Après avoir fini un tour, il convient de vérifier si un joueur a gagné. Pour cela, il faut vérifier s'il existe un alignement de 5 pions en ligne, colonne ou diagonale pour chacun des deux joueurs. Si les deux joueurs ont un alignement de 5 pions alors c'est le joueur dont ce n'était pas le tour qui gagne.

Dans la deuxième partie du sujet, il faudra compter les alignements de n pions consécutifs d'un même joueur, avec $n \leq 5$. On se propose de définir des fonctions intermédiaires qui vont permettre de tester si un alignement de n pions est valide, avec $1 < n \leq 5$:

- `alig(jeu:[[int]], joueur:int, i:int, n:int) -> bool` qui renvoie `True` si un alignement de n pions du joueur passé en argument est trouvé sur la ligne i et `False` sinon ;
- `acol(jeu:[[int]], joueur:int, j:int, n:int) -> bool` qui renvoie `True` si un alignement de n pions du joueur passé en argument est trouvé sur la colonne j et `False` sinon ;
- `adiag1(jeu:[[int]], joueur:int, n:int) -> bool` qui renvoie `True` si un alignement de n pions du joueur passé en argument est trouvé sur la diagonale partant de $(0, 0)$ et `False` sinon ;
- `adiag2(jeu:[[int]], joueur:int, n:int) -> bool` qui renvoie `True` si un alignement de n pions du joueur passé en argument est trouvé sur la diagonale partant de $(0, 4)$ et `False` sinon.

❏ **Q10** – Écrire une fonction
`alig(jeu:[[int]], joueur:int, i:int, n:int) -> bool` telle que décrite précédemment. On veillera à n'accéder qu'une seule fois à la valeur de chaque case dans un souci d'optimalité.

❏ **Q11** – Écrire une fonction `gagnant(jeu:[[int]], joueur:int) -> bool` qui prend en arguments le plateau de jeu à la fin d'un tour et un joueur, qui renvoie `True` si le joueur possède au moins un alignement gagnant et `False` sinon. Cette fonction utilisera les fonctions précédemment définies.

3 Jeu contre un ordinateur

Dans le cadre d'un jeu contre l'ordinateur, il faut définir des fonctions permettant à l'ordinateur de choisir intelligemment un mouvement (choix d'un pion valide et de la nouvelle position). Pour cela, il faut déterminer, dans une configuration de plateau donnée, l'ensemble des mouvements possibles, puis pour chacun d'entre eux en choisir un qui maximise les chances que l'ordinateur a de gagner.

On rappelle que l'on ne peut prendre un pion que sur le bord et à la marque du joueur ou un pion neutre. Pour chaque pion qu'il est possible de prendre, il y a plusieurs choix de nouvelles positions possibles (figure 1(b)).

❑ **Q12** – Dans le cas du plateau en situation initiale, déterminer le nombre exact de mouvements possibles pour le joueur 1.
Déterminer sans justification une situation de jeu où le nombre de mouvements possibles est minimal et donner ce nombre.

Une solution pour définir le choix de l'ordinateur est d'utiliser l'algorithme "minimax".

Considérons l'arbre de jeu donné en exemple sur le cahier réponse. Les carrés correspondent aux tours du joueur MAX et les ronds à ceux du joueur MIN. L'arbre proposé indique 16 configurations atteignables en 4 tours depuis une configuration contrôlée par le joueur MAX. Les valeurs indiquées dans les cases correspondent au score obtenu pour chaque configuration.

❑ **Q13** – Compléter l'arbre en appliquant la stratégie "minimax".

❑ **Q14** – Discuter de la faisabilité de l'utilisation de cet algorithme dans le cadre de ce jeu en basant votre réponse sur le calcul du nombre de mouvements possibles dans le pire des cas sur 3 tours de jeu en partant de la situation initiale.

En pratique, on utilise l'élagage "alphabeta" qui est une variante de l'algorithme "minimax" avec élagage de l'arbre du jeu. Cet algorithme sera détaillé plus loin. On commence par construire les fonctions nécessaires à son fonctionnement.

On définit un mouvement par une liste de 4 éléments représentant les coordonnées de la case de départ (i_d, j_d) puis les nouvelle coordonnées (i_n, j_n) de la case pour repositionner le pion : `[i_d, j_d, i_n, j_n]`.

La fonction `deplacements_possibles(jeu:[[int]], joueur:int) -> [[int]]`, donnée sur le document réponse, prend en arguments le plateau de jeu et un joueur et renvoie la liste des mouvements possibles que le joueur peut faire à son tour.

	0	1	2	3	4
0	O	O	O	X	O
1	O		X		O
2	X	X	X	X	O
3	O	X	X	X	O
4	O	O	O	O	X

FIGURE 3 – Situation de jeu - Le joueur 1 doit jouer

❑ **Q15** – Donner ce que renvoie la fonction `deplacements_possibles` pour le plateau de jeu défini à la figure 3 et pour le joueur 1 en faisant attention à l'ordre des mouvements renvoyés.

Heuristique d'évaluation

L'ordinateur va prévoir son mouvement en anticipant plusieurs tours d'avance. Il va bâtir l'arbre des différentes possibilités de jeux et explorer cet arbre.
L'idéal serait de chercher tous les chemins gagnants mais le nombre de possibilités et le nombre de tours pour les atteindre sont tellement grands que l'arbre est impossible à explorer en totalité. On va se contenter de prévoir quelques tours d'avance et de choisir le meilleur chemin. On ne construit donc l'arbre du jeu que sur quelques niveaux de profondeur.
Le meilleur chemin est défini grâce à une fonction d'évaluation qui renvoie une valeur associée à l'état du plateau de jeu. Si la valeur absolue est très grande et la valeur est positive, alors le joueur 1 est susceptible de gagner. Si la valeur absolue est très grande et la valeur est négative, c'est le joueur 2 qui risque de gagner.

Cette fonction d'évaluation d'une position prend comme arguments : le plateau de jeu à évaluer et la profondeur, qui correspond au nombre de tours restant à évaluer lors de la recherche du meilleur coup possible. Une profondeur nulle correspond à une évaluation directe, une profondeur égale à 1 signifie qu'il y a un tour de jeu après celui-ci, etc.

La fonction d'évaluation construit un score positif pour le joueur 1 et négatif pour le joueur 2. La fonction suit les règles suivantes :

- si le plateau est gagnant alors on renvoie `100 + profondeur` pour le joueur 1 et `-100 - profondeur` pour le joueur 2. La valeur 100 est conventionnelle. Elle est choisie uniquement pour favoriser les branches gagnantes.
- sinon, on construit une valeur en fonction du joueur considéré :
  - en ajoutant 5 fois le nombre d'alignements de 4 pions du joueur ;
  - en ajoutant 20 si la case centrale appartient au joueur ;
  - en ajoutant le nombre de pions du joueur et en soustrayant le nombre de pions de son adversaire ;
  - en ajoutant la profondeur ;
  - la fonction renvoie la valeur si le joueur considéré est le joueur 1 et l'opposé de cette valeur sinon.

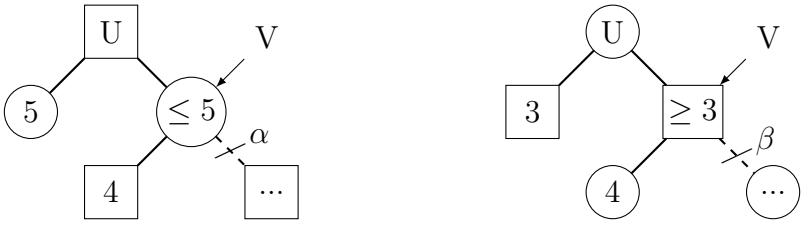
❑ **Q16** – Écrire une fonction `alignement_4(jeu:[[int]], joueur:int) -> int` qui prend en arguments le plateau de jeu et un joueur et qui renvoie le nombre d'alignements de 4 pions de ce joueur. Cette fonction utilisera les fonctions `alig`, `acol`... définies avant la question 10.

❑ **Q17** – Écrire une fonction :
`evaluation(jeu:[[int]], joueur:int, profondeur:int) -> int` qui prend en arguments le plateau de jeu, le joueur et la profondeur de la recherche et qui renvoie le résultat de l'évaluation d'une position du jeu.

Élagage "alphabeta"

L'algorithme "alphabeta" est fondé sur l'algorithme "minimax" mais, au lieu de parcourir entièrement l'arbre de jeu, des simplifications sont faites en n'explorant pas toutes les branches ; on parle d'élagage.

Illustrons le principe sur des arbres simples en utilisant la même convention que précédemment : les nœuds MAX sont représentés par des carrés et les nœuds MIN sont représentés par des ronds.



(a) Coupure alpha (b) Coupure beta

FIGURE 4 – Illustration de l'élagage avec les coupures alpha et beta.

L'élagage alpha est illustré sur l'exemple de la figure 4(a). Pour réaliser l'évaluation du nœud MAX appelé U, on va prendre le maximum des nœuds MIN enfants. Le premier enfant donne une valeur de 5, ainsi la valeur de U sera au moins de 5.
Supposons que le premier enfant de V donne une valeur de 4 (inférieure à 5), cela ne sert à rien de poursuivre l'évaluation des autres branches de V car si les valeurs sont plus grandes que 4, le joueur MIN choisira la plus petite valeur (donc 4) et comme cette valeur est inférieure à 5, ce sera la valeur 5 qui remontera au niveau de U.

L'élagage beta est illustré sur l'exemple de la figure 4(b). Pour réaliser l'évaluation du nœud MIN noté U, on va choisir le minimum des nœuds MAX enfants. Le premier enfant donne une valeur de 3, ainsi la valeur de U sera au plus 3 (car le joueur MIN choisira la valeur la plus petite).
Si le premier enfant de V a une valeur de 4, alors la valeur de V sera au moins 4.
La valeur de V sera donc supérieure à 3, il ne sert à rien de poursuivre l'évaluation des autres enfants de V (car le joueur MIN prendra la valeur la plus petite donc 3).

❑ **Q18** – En reprenant l'exemple de la question 13, compléter les nœuds qu'il faut déterminer et représenter les coupures des branches non calculées, comme sur la figure 4, en supposant que la construction se fait toujours en commençant par les nœuds situés à gauche.

On donne le pseudo-code de l'algorithme "alphabeta" avec une profondeur maximale d'exploration donnée qui calcule la valeur associée à un noeud :

```
alphabeta(noeud, alpha, beta, profondeur)
  si noeud est une feuille ou profondeur atteinte alors
      renvoyer la valeur de l'heuristique du noeud
  profondeur = profondeur - 1
  si noeud de type Max alors
      v = -infini
      pour tout fils de noeud faire
          v = max(v, alphabeta(fils, alpha, beta, profondeur))
          si v >= beta alors #coupure beta
              renvoyer v
          alpha = max(alpha, v)
  sinon
      v = infini
      pour tout fils de noeud faire
          v = min(v, alphabeta(fils, alpha, beta, profondeur))
          si alpha >= v alors #coupure alpha
              renvoyer v
          beta = min(beta, v)
  renvoyer v
```

L'implémentation de ce pseudo-code est donnée sur l'ébauche suivante. La fonction renvoie la valeur associée au nœud étudié ainsi que le déplacement qui a permis d'atteindre cette valeur :

```
1  def alphabeta(jeu:[[int]], joueur:int, alpha:float, \
2                                         beta: float, profondeur:int) -> tuple:
3      if  ..................................... : #à compléter
4          .....................................
5          return eval, None
6      profondeur = profondeur - 1
7      meilleur_depl = None
8      if joueur == 1: #noeud Max
9          v = -math.inf
10         for depl in deplacements_possibles(jeu, joueur):
11             jeu_c = copy.deepcopy(jeu)
12             deplacement_case(.....................................)
13             v = max(v, alphabeta(.....................................)[0])
14             if v >= beta:
15                 return v, meilleur_depl
16             if v > alpha:
17                 alpha = v
18                 meilleur_depl = depl
19     else: #noeud Min
20         v = math.inf
21         for depl in deplacements_possibles(jeu, joueur):
22             jeu_c = copy.deepcopy(jeu)
23             deplacement_case(.........NON DEMANDE.................)
24             v = min(v, alphabeta(............NON DEMANDE.................)[0])
25             if alpha >= v:
26                 return v, depl
27             if v < beta:
28                 beta = v
29                 meilleur_depl = depl
30     return v, meilleur_depl
```

❑ **Q19** – Compléter, à l'aide du pseudo-code, les lignes 3, 4, 12 et 13 de la fonction `alphabeta`.

4 Gestion d'une base de données

Pour réaliser des analyses sur les parties, on les stocke dans une base de données. Cette base de données est composée de 2 tables :

- **Partie** avec les attributs :
  - `id` l'identifiant d'une partie, clé primaire, de type entier ;
  - `id_joueur1` l'identifiant du joueur 1, clé étrangère, de type entier ;
  - `id_joueur2` l'identifiant du joueur 2, clé étrangère, de type entier ;
  - `id_gagnant` l'identifiant du gagnant, clé étrangère, de type entier ;
  - `date` la date de la partie, de type chaîne de caractères ;
  - `nbtours` le nombre de tours de la partie, de type entier.
- **Joueur** avec les attributs :
  - `id` l'identifiant du joueur, clé primaire, de type entier ;
  - `nom` le nom du joueur, de type chaîne de caractères ;
  - `prenom` le prénom du joueur, de type chaîne de caractères.

❑ **Q20** – Écrire une requête SQL qui renvoie le nombre de parties et le nombre moyen de tours réalisés par le joueur nommé "John Doe" quand il a commencé la partie (il est le joueur 1).

❑ **Q21** – Écrire une requête SQL qui renvoie le nombre de tours de la partie ayant nécessité le moins de tours, ainsi que les noms et prénoms des joueurs 1 et 2 ayant participé à cette partie. On supposera l'unicité de cette partie.

Fin de l'épreuve.

Numéro d'inscription

Né(e) le / /

Nom

Prénom (s)

Signature

Épreuve : .....

Les feuilles dont l'entête d'identification n'est pas entièrement renseignée ne seront pas prise en compte pour la correction.

Feuille /

Cahier Réponse - IC

Question 1

Question 2

Question 3

NE RIEN ÉCRIRE

DANS CE CADRE

Question 4

Question 5

Question 6

Question 7

```
def deplacement_case(jeu:[[int]], i_d:int, j_d:int, i_n:int, j_n:int, joueur:int)->None:
    ''' jeu : liste de listes réprésentant le plateau de jeu
        i_d, j_d : coordonnées du pion au départ
        i_n, j_n : nouvelles coordonnées du pion
        joueur : numéro du joueur dont c'est le tour'''
    if i_n < i_d: #mouvement 1
        i = i_d
        while i > i_n:
            jeu[i][j_n] = jeu[i-1][j_n]
            i = i - 1
        jeu[i_n][j_n] = joueur
    elif i_n > i_d: #mouvement 2
        i = i_d
        while i < i_n:
            jeu[i][j_n] = ............
            i = ............
        jeu[i_n][j_n] = joueur
    elif j_n < j_d: #mouvement 3
        j = j_d
        while j > j_n:
            jeu[i_n][j] = jeu[i_n][j-1]
            j = j - 1
        jeu[i_n][j_n] = joueur
    else: #mouvement 4
        j = j_d
        while j < j_n:
            jeu[i_n][j] = jeu[i_n][j+1]
            j = j + 1
        jeu[i_n][j_n] = joueur
```

Question 10

Question 11

Numéro d'inscription

Né(e) le / /

Signature

Nom

Prénom (s)

Concours commun Mines-Ponts

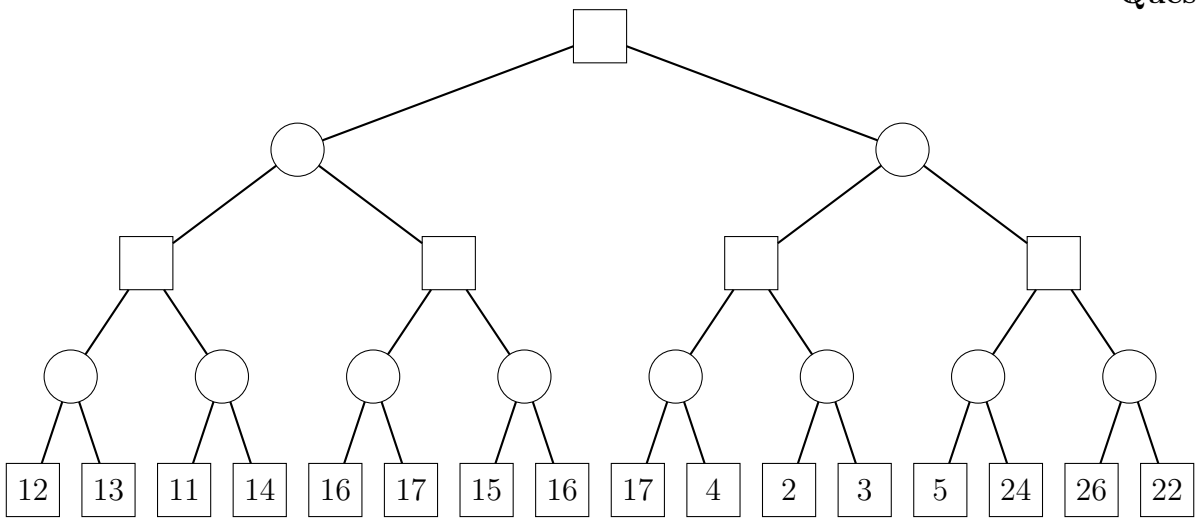
Épreuve : .....

Les feuilles dont l'entête d'identification n'est pas entièrement renseignée ne seront pas prise en compte pour la correction.

Feuille /

Question 12

Question 13



Question 14

NE RIEN ÉCRIRE

DANS CE CADRE

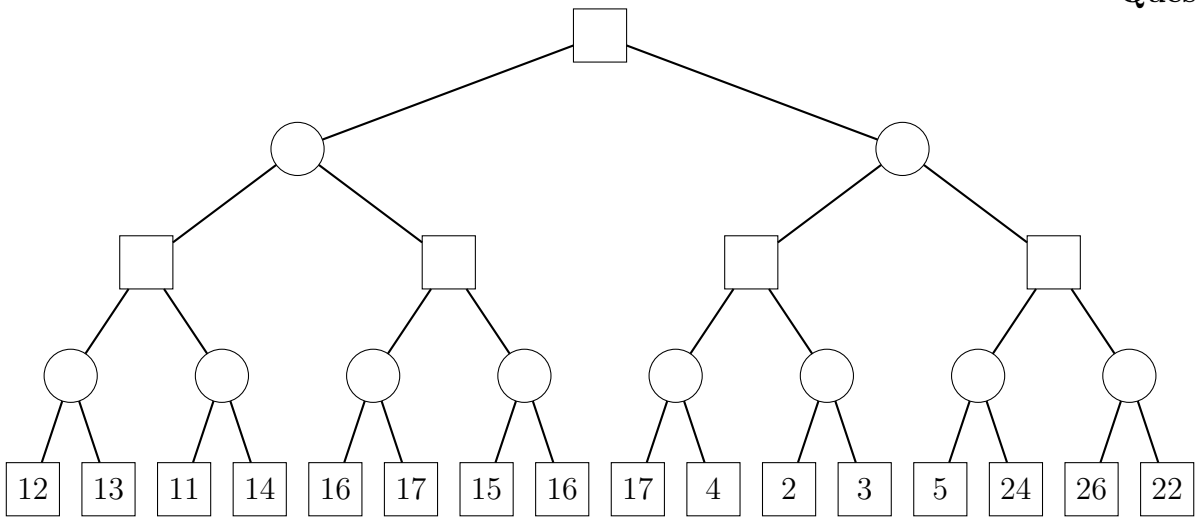
Question 15

```
def deplacements_possibles(jeu:[[int]], joueur:int) -> list:
    '''Un mouvement possible est composé des coordonnées de la case de départ
    et des coordonnées de la case d'arrivée'''
    mvt = []
    for j in range(5): #on traite la première et la dernière ligne
        if jeu[0][j] in [0,joueur]: #case (0,j) OK
            if j != 0:
                mvt.append([0,j,0,0])
            if j != 4:
                mvt.append([0,j,0,4])
            mvt.append([0,j,4,j])
        if jeu[4][j] in [0,joueur]: #case (4,j) OK
            if j != 0:
                mvt.append([4,j,4,0])
            if j != 4:
                mvt.append([4,j,4,4])
            mvt.append([4,j,0,j])
    for i in range(1,4):
        if jeu[i][0] in [0, joueur]: #première colonne, angle exclu
            mvt.append([i,0,i,4])
            mvt.append([i,0,0,0])
            mvt.append([i,0,4,0])
        if jeu[i][4] in [0, joueur]: #dernière colonne, angle exclu
            mvt.append([i,4,i,0])
            mvt.append([i,4,0,4])
            mvt.append([i,4,4,4])
    return mvt
```

Question 16

Question 17

Question 18



Question 19

ligne 3 :

ligne 4 :

ligne 12 :

ligne 13 :

Question 20

Question 21