Concours e3a-Polytech, France

# Session 2024, filière PSI

## Informatique

Durée : 3 heures

### Ce que traite le sujet

- Partie I : présentation et règles
- Partie II : programmation de la structure de jeu
- Partie III : programmation de l'intelligence artificielle

**Sujet reproduit sans retouche à partir de la page suivante.**

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/e3a/psi/2024/info

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

# ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

# ÉPREUVE SPÉCIFIQUE - FILIÈRE PSI

---

# INFORMATIQUE

**Durée : 3 heures**

---

*N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.*

***RAPPEL DES CONSIGNES***

- *Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.*
- *Ne pas utiliser de correcteur.*
- *Écrire le mot FIN à la fin de votre composition.*

---

**Les calculatrices sont interdites.**

**Le sujet est composé de trois parties.**

L'épreuve est à traiter en langage **Python** sauf pour les bases de données.
Les différents algorithmes doivent être rendus dans leur forme définitive sur le **Document Réponse** dans l'espace réservé à cet effet en respectant les éléments de syntaxe du langage (les brouillons ne sont pas acceptés).
La réponse ne doit pas se limiter à la rédaction de l'algorithme sans explication, les programmes doivent être expliqués et commentés de manière raisonnable.

Énoncé et Annexe : 16 pages
Document Réponse : 12 pages

**Seul le Document Réponse doit être rendu dans son intégralité (le QR Code doit être collé sur la première page de ce Document Réponse).**

# Le jeu de l'awalé

Les fonctions seront définies avec leur signature dans le sujet :

**ma_fonction(arg1:type1, arg2:type2) → type3**.

Cette notation permet de définir une fonction qui se nomme **ma_fonction** qui prend deux arguments en entrée **arg1** de type **type1** et **arg2** de type **type2**. Cette fonction renvoie une valeur de type **type3**.
Il ne faut pas recopier les signatures des fonctions dans le **Document Réponse (DR)**, il faut écrire directement :

```
def ma_fonction(arg1, arg2):
    # liste d'instructions
```

## Partie I - Présentation et règles

Le sujet porte sur l'étude de l'awalé, un jeu de stratégie très ancien qui fait partie des jeux de semailles. En effet, à son origine, il était pratiqué à l'aide de graines qui étaient semées dans deux rangées de 6 trous creusés dans un plateau en bois ou à même le sol. Ce jeu est très répandu en Afrique et à partir du XVII$^{e}$ siècle des indices de sa pratique sont également trouvés en Amérique du Sud et en Asie.

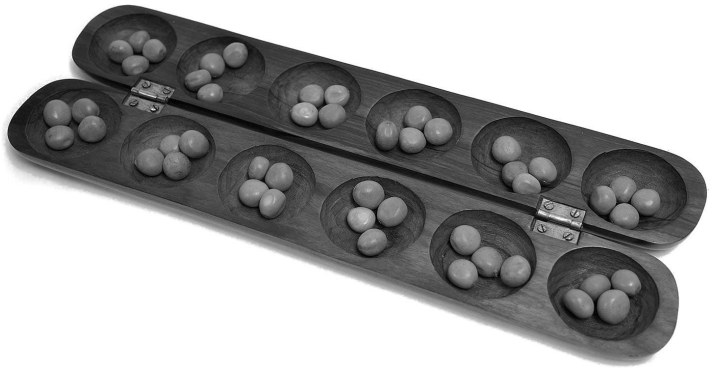


**Figure 1 -** Plateau d'awalé en bois

Les règles de ce jeu sont particulièrement simples et s'apprennent rapidement. Il en existe plusieurs variantes mais ce sujet n'en exposera qu'une seule. L'awalé se joue à deux. À tour de rôle, les joueurs prennent les graines situées dans un trou de leur rangée pour ensuite les déplacer dans les autres trous. Des graines peuvent ensuite être récoltées pour que les joueurs se constituent une réserve personnelle.

### I.1 - But du jeu

L'objectif est de récolter plus de graines que son adversaire. Au départ, le plateau est composé de 2 rangées de 6 trous contenant chacun 4 graines comme le montre la **figure 2**. Le jeu s'arrête si l'un des joueurs obtient dans sa réserve personnelle à côté du plateau plus de la moitié des graines en jeu, c'est-à-dire au moins 25 graines ou jusqu'à une situation empêchant le gain de nouvelles graines.

### I.2 - Déroulement de la partie

Les 2 participants jouent à tour de rôle. Les joueurs sont appelés par la suite Alice et Bob ; Alice joue en premier. Les joueurs sont face à face. La rangée de 6 trous située juste devant le joueur est appelée son camp.

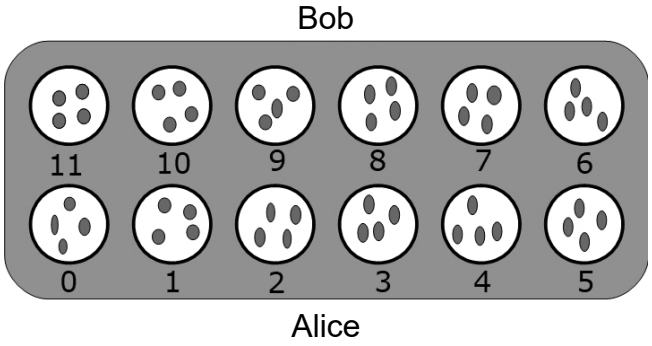



**Figure 2 -** Situation de départ

Chaque coup consiste à choisir une case non vide de son camp, à prendre toutes les graines de cette case en main et à les semer à raison d'une graine par case en suivant le sens direct, c'est-à-dire le sens inverse des aiguilles d'une montre (**figure 3**). On ne sème jamais de graine dans la case d'origine choisie. En effet, si la case de départ choisie contient plus de 11 graines, le joueur va semer sur un tour de plateau complet et revenir à la case d'origine des graines ; il faut alors sauter cette case pour continuer de semer les graines dans les autres cases. À la fin de son tour de jeu, la case de départ choisie par le joueur est nécessairement vide.

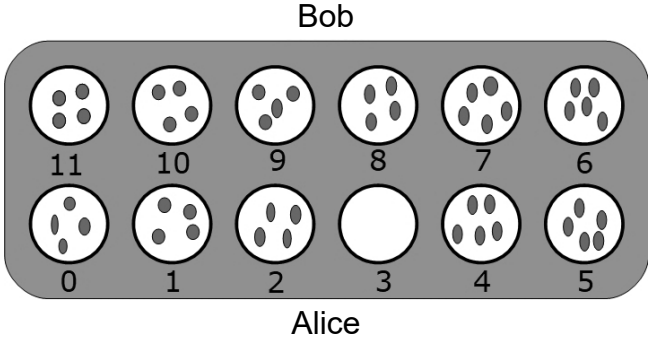



**Figure 3 -** Situation obtenue après qu'Alice ait semé la case d'indice 3

Une règle fondamentale de l'awalé est l'interdiction d'affamer son adversaire (**règle de la famine**). Lorsqu'un joueur n'a plus de graines dans son camp, son adversaire est obligé de jouer un coup qui lui en apporte au moins une.
Par ailleurs, il est interdit de jouer un coup qui ôte, après récolte, toutes les graines du camp adverse.

Une fois qu'un joueur a terminé de semer, il peut récolter les graines du plateau de jeu (en respectant la règle précédente). La récolte consiste à retirer les graines du plateau pour les stocker dans sa réserve personnelle sur la table à côté du plateau de jeu. Le joueur récolte les graines disponibles après son tour de semence, en commençant par la dernière case dans laquelle il a semé et sous les conditions suivantes :

- la case appartient au camp adverse (condition 1) ;
- cette case contient exactement 2 ou 3 graines (condition 2) ;
- s'il vient de ramasser les graines de la case, le joueur doit continuer la récolte dans le sens inverse de la semence, si la case respecte les deux premières conditions ;
- il est interdit d'affamer son adversaire, on ne peut donc pas prendre toutes les graines du camp adverse. Si la phase de récolte se termine ainsi, alors la récolte est annulée (condition 3 liée à la règle de la famine).

### I.3 - Conditions de fin

La partie s'arrête sous certaines conditions :

- un joueur obtient au moins 25 graines dans sa réserve ;
- 3 graines ou moins restent sur le plateau ;
- un joueur est dans l'incapacité de jouer car aucun coup ne permet de respecter les différentes règles.

À la fin de la partie, chaque joueur ramasse les graines de son camp pour les transférer dans sa réserve personnelle. Le décompte des points peut alors se faire. Le joueur ayant obtenu au moins 25 graines est alors déclaré vainqueur. Une situation de nullité est possible si les deux joueurs obtiennent autant de graines chacun à la fin de la partie.

### I.4 - Compréhension des règles

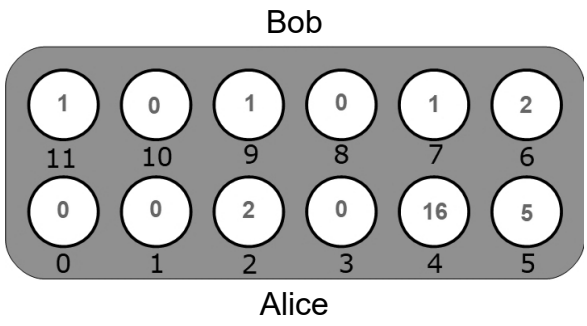



**Figure 4 -** Exemple de situation - Alice doit jouer

On considère la situation de jeu indiquée par la **figure 4**. C'est au tour d'Alice de jouer.

**Q1.** Indiquer, en justifiant, les indices des cases qu'Alice peut choisir de jouer.

**Q2.** Sur le **DR**, pour chacun des choix de cases possibles, renseigner la situation possible du plateau de jeu après le coup d'Alice (c'est-à-dire après avoir semé et récolté les graines). Indiquer également le gain éventuel pour chaque coup possible de la **figure 4**.

On considère les deux situations de jeu indiquées par la **figure 5**. C'est au tour d'Alice de jouer.

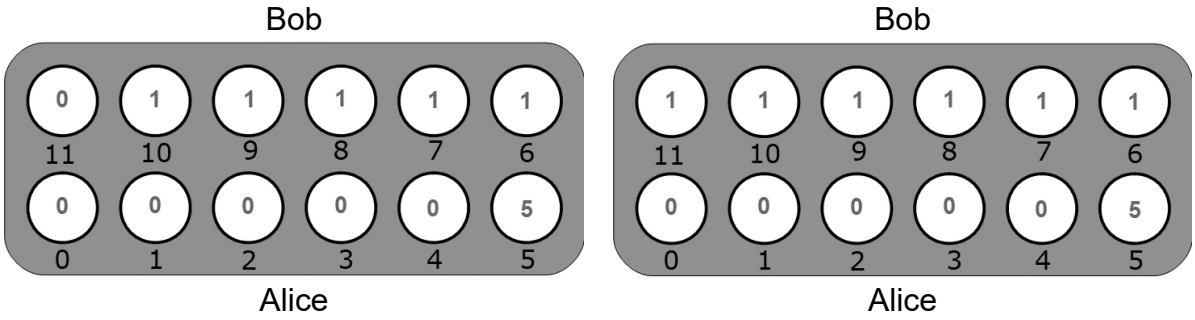



**Figure 5 -** Autres exemples de situation - Alice doit jouer

**Q3.** Pour chacune des deux situations de jeu de la **figure 5**, donner et justifier la situation du plateau après le tour de jeu d'Alice.

# Partie II - Programmation de la structure de jeu

Cette partie aborde la modélisation et la structure du jeu dans le langage Python.

## II.1 - Représentation du jeu

Le choix d'un dictionnaire a été fait pour stocker l'ensemble des données du jeu. De plus, l'appel de la fonction **intialisation(nom_joueur1:str,nom_joueur2:str) → dict** permet de créer la structure du jeu dans les conditions de départ.

```
def initialisation(nom_joueur1, nom_joueur2) :
    jeu = {}
    jeu['joueur1'] = nom_joueur1 # Nom du premier joueur
    jeu['joueur2'] = nom_joueur2 # Nom du second joueur
    jeu['score'] = [0,0]         # Réserve du joueur1, puis du joueur2
    jeu['n'] = 0                 # Nombre de tours déjà effectués
    jeu['plateau'] = [4]*12      # Plateau de jeu initial
    return jeu
```

Le compteur de tours détermine le tour de jeu des joueurs et commence donc à zéro. Le joueur1 commence la partie.
Le plateau de jeu est séparé en deux parties égales. Les six premières cases correspondent à celles du joueur dont c'est le tour ; les six dernières cases sont celles de l'adversaire. À la fin d'un tour de jeu, il faut échanger les deux ensembles de six cases. Ainsi les cases d'indices 0 à 5 correspondent toujours à celles du joueur dont c'est le tour (joueur actif) et les cases d'indices 6 à 11 à celles de son adversaire.
L'argument **jeu:dict** fait référence à un dictionnaire représentant le jeu de structure identique à celui renvoyé par **initialisation**.

**Q4.** Donner la parité de **jeu['n']** lorsque c'est au tour du joueur1 de jouer. Écrire une fonction **tour_joueur1(jeu:dict) → bool** qui renvoie **True** si c'est le tour de jeu du joueur1 et **False** sinon.

**Q5.** Écrire une fonction **tourner_plateau(jeu:dict) → None** qui modifie l'entrée **"plateau"** du dictionnaire **jeu** en échangeant les cases d'indices 0 à 5 avec celles de 6 à 11 (**figure 6**).

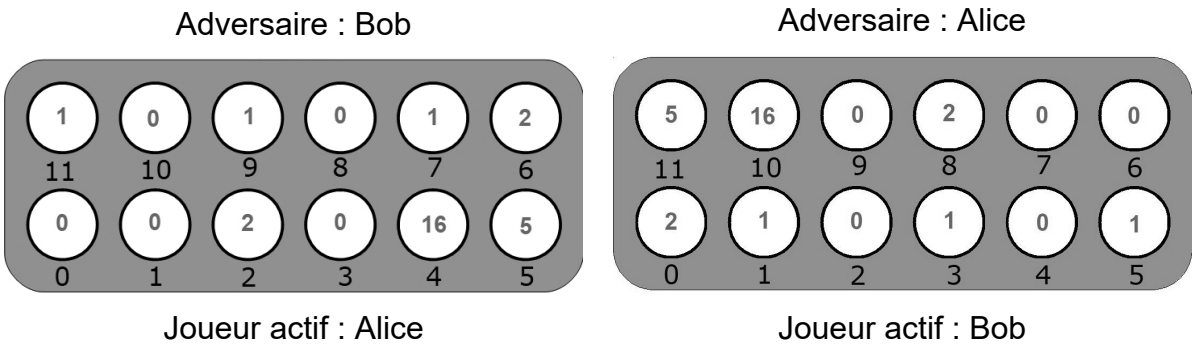



**Figure 6 -** Exemple d'inversion de plateau à la fin d'un tour entre Alice et Bob

**Q6.** Donner le maximum de graines que peut contenir une case. Déterminer alors le nombre de bits nécessaires pour coder les entiers représentants le nombre de graines par case.

Dans la suite du sujet, nous aurons besoin d'une fonction permettant de copier le jeu (dictionnaire) en entier dont certains éléments sont des listes.

**Q7.** Écrire une fonction **copie(jeu:dict) → dict** qui renvoie une copie profonde de la structure de dictionnaire retenue. Il est interdit d'utiliser la fonction **deepcopy** du module **copy**. L'opérateur **copy** des listes est autorisé.

Le déroulement d'un jeu d'awalé entre deux joueurs "humains" a la structure suivante :

```
1 def awale_jcj(nom_joueur1, nom_joueur2):
2     jeu = initialisation(nom_joueur1, nom_joueur2)
3     jeu_continue = True
4     while jeu_continue :
5         affiche(jeu['plateau'])
6         case_choisie = int(input("Choisir une case : "))
7         jeu_continue = tour_jeu(jeu, case_choisie)
8     return gagnant(jeu)
```

Remarque : la fonction **input** permet de récupérer une chaîne de caractères fournie par le joueur dans la console.
La fonction **affiche** permet d'afficher le plateau de jeu à l'écran.
Dans le code proposé, nous considérons que le joueur ne commet pas d'erreur de frappe et rentre toujours dans la console l'indice de la case choisie.

### II.2 - Programmation d'un tour de jeu

Une fois la case de début de tour choisie par le joueur, un tour de jeu a la structure suivante :

1. tester si la case où l'on prend les graines est valide ou non (les conditions de validité sont explicitées ensuite) ;
2. si le choix est valide, semer les graines puis récolter des graines. On incrémente alors le nombre de tours, on ajoute au score du joueur le nombre de graines récoltées et on échange les deux parties du plateau ;
3. tester si la partie est finie ou non (les conditions sont décrites dans la première partie).

**Les pré-conditions des fonctions de cette partie et de la partie suivante sont répertoriées ci-dessous :**

- l'argument **jeu:dict** fait référence à un dictionnaire représentant le jeu comme présenté au début de cette partie ;
- l'argument **plateau:[int]** fait référence à une structure de données similaire à ce que contient **jeu['plateau']** ;
- l'argument **case:int** fait référence à un entier compris entre 0 inclus et 12 exclu.

On s'intéresse tout d'abord à l'étape 2 de manière à écrire deux fonctions :

- **deplacer_graines(plateau:[int], case:int) → int** ;
- **ramasser_graines(plateau:[int], case:int) → int**.

La fonction **deplacer_graines(plateau:[int], case:int) → int** prend comme argument le plateau de jeu ayant la structure choisie précédemment et la case non vide où le joueur actuel prend les graines. Cette fonction réalise la prise des graines de la **case** choisie et les sème une par une dans le sens direct (sens inverse des aiguilles d'une montre). Elle renvoie l'indice de la case où la dernière graine a été semée.

**Q8.** Proposer une fonction **deplacer_graines(plateau:[int],case:int) → int** modifiant en place l'argument **plateau**. On rappelle que l'on ne sème pas de graine dans la case choisie en début du tour.

**Q9.** Écrire une fonction auxiliaire **case_ramassable(plateau:[int], case:int) → bool** qui renverra **True** si le joueur dont c'est le tour a le droit de ramasser les graines de la case proposée et **False** sinon. On ne testera pas la question de la famine pour simplifier le problème. Pour rappel, le joueur peut ramasser le contenu de la case si :

- la case appartient au camp de l'adversaire, soit toujours dans la deuxième moitié du plateau ;
- la case contient 2 ou 3 graines.

La fonction **ramasser_graines(plateau:[int], case:int) → int** prend comme argument le plateau de jeu après déplacement des graines et l'indice de la case où la dernière graine a été semée. Cette fonction procède au ramassage des graines.

Si le joueur peut ramasser le contenu de la case, alors il ramasse le contenu et passe à la case précédente puis ramasse les graines de cette case si ces mêmes conditions sont respectées et ainsi de suite. Pour simplifier, la fonction **ramasser_graines** ne testera pas la condition de famine citée précédemment. Elle renverra le nombre de graines récoltées (c'est-à-dire le nombre de points gagnés).

**Q10.** Écrire la fonction **ramasser_graines(plateau:[int], case:int) → int**. On demande que la fonction **ramasser_graines** soit une fonction récursive. Cette fonction utilisera la fonction précédente **case_ramassable** et devra modifier le plateau en place et renvoyer le résultat de la récolte des graines.

Il faut vérifier à chaque tour de jeu si le choix d'une case est autorisé ou non. Si c'est un joueur humain qui joue, son choix peut se porter sur une case "interdite", c'est-à-dire dont il ne peut pas prendre les graines. Si c'est un joueur virtuel, celui-ci doit pouvoir faire la liste des cases "acceptables". Une case est "acceptable" si :

- condition 1 : elle est du côté du joueur dont c'est le tour ;
- condition 2 : elle est non vide ;
- condition 3 : à la fin du tour de jeu, les cases de l'adversaire ne sont pas complètement vides (condition de famine).

**Q11.** Écrire une fonction **test_famine(plateau:[int], case:int) → bool** qui vérifie que la case **case** choisie vérifie la condition 3 (renvoie **True** si la condition 3 est vérifiée et **False** sinon). Il n'est pas nécessaire qu'elle vérifie les deux conditions 1 et 2. Il est possible d'utiliser les fonctions demandées précédemment quelle que soit leur implémentation. *Remarque : les arguments d'entrée ne doivent pas être modifiés par la fonction.*

**Q12.** On propose ci-dessous une fonction qui vérifie que la case choisie vérifie bien les trois conditions précédentes, connaissant l'état actuel du jeu (le plateau). Compléter la condition de valeur **test** afin de déterminer si la case est acceptable ou non.

```
1 def test_case(plateau, case):
2     """ Vérifie si la case choisie par le joueur est acceptable
3     renvoie True si la case est acceptable, False sinon """
4     condition3 = test_famine(plateau, case)
5     # Case acceptable
6     test =  # à compléter
7     return test
```

**Q13.** Écrire la fonction **cases_possibles(jeu:dict) → [int]** qui renvoie la liste des indices de toutes les cases jouables par le joueur actif. Le dictionnaire **jeu** ainsi que sa clé **plateau** ne devront pas être modifiés.

Après un tour, il faut vérifier si le jeu est terminé ou si les joueurs peuvent continuer à jouer. Le jeu se termine si l'une des conditions suivantes est vérifiée :

- un des joueurs possède plus de la moitié des graines (soit au moins 25) ;
- le nombre de tours joués est supérieur ou égal à 100 (pour éviter un jeu infini lorsqu'il y a peu de graines) ;
- il reste 3 graines ou moins sur le plateau ;
- le joueur qui va jouer ne possède plus de case jouable. On suppose que le plateau a été échangé avant de faire les tests, donc le joueur qui doit jouer a ses graines dans les cases d'indices 0 à 5.

**Q14.** Écrire une fonction **tour_suivant(jeu:dict) → bool**. Cette fonction renvoie **True** si le jeu peut continuer et **False** sinon.

L'ébauche de la fonction **tour_jeu(jeu:dict, case:int) → bool** est donnée ci-dessous. Si le test de la case n'est pas valide, la fonction affiche un message et renvoie **True** pour permettre au joueur de proposer une nouvelle case. Sinon, elle renvoie **True** si le jeu peut continuer et **False** si le jeu ne peut pas continuer.

**Q15.** Donner l'instruction 1, l'instruction 2, la condition 1 et l'instruction 3 permettant à la fonction de répondre à la description précédente.

```
def tour_jeu(jeu, case):
    plateau = jeu['plateau']
    if test_case(plateau, case):  # La case jouée est acceptable
        # Instruction 1 : deplacer les graines
        # Instruction 2 : ramasser les graines
        """Pour augmenter le score, il faut savoir qui joue grâce à la
           parité du nombre de tours"""
        if # Condition 1:
            jeu['score'][0] = jeu['score'][0] + graines_gagnees
        else:
            jeu['score'][1] = jeu['score'][1] + graines_gagnees
        # Instruction 3  # On incrémente le nombre de tours
        tourner_plateau(jeu)  # Echanger les plateaux
        return tour_suivant(jeu)
    else:
        print("La case choisie n'est pas valable")
        return True
```

**Q16.** Écrire la fonction **gagnant(jeu:dict) → str** prenant comme argument le dictionnaire **jeu** contenant l'état actuel du jeu. La fonction doit procéder au ramassage des graines de chaque côté du plateau et les affecter au score, puis renvoyer le nom du joueur qui a gagné, c'est-à-dire qui a le plus de graines à la fin du jeu. S'il y a match nul, la fonction devra renvoyer la chaîne de caractère **"égalité"**.

# Partie III - Programmation de l'Intelligence Artificielle (IA)

Cette partie aborde la programmation de l'Intelligence Artificielle (IA) si l'on désire jouer contre l'ordinateur. La structure du déroulement du jeu reste la même que précédemment, seul le choix de la case de jeu est différent. Il s'agit ici de programmer l'IA de manière à ce qu'elle choisisse la meilleure case pour elle. Nous allons pour cela utiliser un algorithme MinMax ou plus précisément sa version appelée Negamax.

## III.1 - Arbre des configurations

On peut décrire l'ensemble des configurations possibles du plateau par un arbre orienté où chaque nœud correspond à un état du jeu (il peut donc être représenté par le dictionnaire **jeu**). Chaque arête orientée correspond à un tour de jeu. Un arbre est un graphe qui ne possède pas de cycle.

On rappelle le rôle des fonctions utiles créées dans les parties précédentes :

- **copie(jeu:dict) → dict** : réalise la copie profonde du dictionnaire **jeu** passé en argument ;
- **deplacer_graines(plateau:[int], case:int) → int** : réalise le déplacement des graines sur le **plateau** depuis la **case** choisie et renvoie la case (entier) où la dernière graine a été déposée ;
- **ramasser_graines(plateau:[int], case:int) → int** : réalise le ramassage des graines sur le **plateau** depuis la **case** où la dernière graine a été déposée et renvoie le nombre de graines récoltées (nombre de points gagnés) ;
- **tour_suivant(jeu:dict) → bool** : teste si la configuration de jeu permet de continuer, renvoie **True** si c'est le cas et **False** sinon ;
- **test_case(plateau:[int], case:int) → bool** : teste si le joueur dont c'est le tour a le droit de jouer la **case** choisie, renvoie **True** si c'est le cas et **False** sinon ;
- **cases_possibles(jeu:dict) → [int]** : renvoie la liste des cases jouables par le joueur actif.

Le grand nombre de possibilités de coups rend impossible la description complète de l'arbre. Pour choisir le coup à jouer, l'IA ne va parcourir l'arbre que sur une profondeur choisie à l'avance à partir de la configuration de jeu au moment où c'est à elle de jouer. Lors du parcours, il est important de déterminer trois caractéristiques :

- est-ce que le nœud est une feuille, c'est-à-dire une configuration où le jeu se termine ? La fonction **tour_suivant(jeu:dict)** écrite dans la partie précédente sert à cet effet ;
- combien d'enfants possèdent un nœud, c'est-à-dire, quels coups sont possibles à partir d'une configuration de jeu ?
- quel est le nombre de graines gagnées quand on passe d'un nœud à un autre ?

**Q17.** Grâce aux fonctions proposées précédemment et en s'inspirant de la fonction **tour_jeu**, écrire une fonction **gain(jeu:dict, case:int) → int,dict** qui renvoie le nombre de graines gagnées et un **nouveau dictionnaire** contenant l'état du jeu après le coup (toutes les grandeurs seront mises à jour). On considère que la case est valide, ce n'est pas la peine de le vérifier ici.
*Remarque : les arguments d'entrée ne doivent pas être modifiés par la fonction.*

### III.2 - Algorithme MinMax

L'algorithme MinMax est un algorithme de théorie des jeux consistant à minimiser la perte maximale pour des jeux à deux joueurs. Le principe de l'algorithme est de visiter l'arbre sur une profondeur donnée et de remonter une "valeur de jeu" estimée pour chaque coup possible par une fonction d'utilité. Pour un nœud donné :

- si c'est au joueur dont c'est le tour de jeu, on remonte la valeur de jeu maximale (la plus favorable au joueur) ;
- si c'est à l'adversaire dont c'est le tour de jeu, on remonte la valeur de jeu minimale (la plus favorable à l'adversaire).

La valeur de jeu est calculée récursivement à partir des valeurs de jeu remontées depuis les nœuds enfants. Le calcul dépend du jeu considéré.

Le jeu d'awalé se prête particulièrement bien à une variante de l'algorithme appelée Negamax car la façon d'évaluer la valeur de jeu est symétrique par rapport à 0 entre le joueur et l'adversaire. Ainsi, au lieu de différencier le cas "Joueur" et "Adversaire", il suffit à chaque nœud $p$ dont les enfants sont notés $p_i$ de remarquer que :

$$\min_i(NegaMax(p_i)) = \max_i(-NegaMax(p_i)).$$

#### Comment est estimée la valeur de jeu dans le cas de l'awalé ?

Le cas de l'awalé est assez simple car la valeur de jeu est assez évidente : il s'agit de la différence de graines gagnées par chaque camp, positive si le joueur en ramasse plus et négative si c'est l'adversaire. Pour la calculer, on procéde de cette manière :

- si le nœud est une feuille, on connaît alors qui est le gagnant. On va donc tester qui est le gagnant et :
  - si c'est le joueur actif, on renvoie une valeur de jeu très grande (500) qui ne pourra être dépassée que par une autre configuration gagnante ;
  - si c'est l'adversaire, on renvoie une valeur de jeu très petite ($-500$) qui sera forcément dépassée par une autre configuration non perdante ;
- si l'on a atteint la profondeur maximale fixée, il n'y a pas de gain supplémentaire, donc cette valeur est nulle $NegaAwale(p_{term}) = 0$ ;
- sinon, pour chaque nœud enfant $p_i$, on calcule le gain $g_i$ (nombre de graines ramassées) pour passer du nœud $p$ au nœud $p_i$, puis on **lui retranche** la valeur de jeu calculée au nœud $p_i$ . La valeur de jeu renvoyée par $NegaAwale(p)$ correspond alors au maximum des différences, soit $NegaAwale(p) = \max_i(g_i - NegaAwale(p_i))$.

Pour la question suivante, on choisit une profondeur de 2 et pour faciliter la lecture du plateau, **le plateau n'a pas été inversé entre deux tours de jeu**. On donne dans le **DR** un arbre indiquant la valeur du gain G obtenue en passant d'un nœud supérieur à un nœud inférieur, puis la valeur de jeu VJ du nœud inférieur, égale à $NegaAwale(p)$. On suppose que, pour le sommet (a), Bob vient tout juste de jouer.

À titre d'exemple, la branche de gauche, où Alice joue sa deuxième case, a été remplie :

- en appliquant les règles de jeu, on obtient facilement que le gain de (a) vers (b) vaut 4, le gain de (b) vers (c) vaut 4 et le gain de (b) vers (d) vaut 0 ;
- les valeurs de jeu de (c) et (d) sont nulles car on a atteint la profondeur de 2 ;
- enfin la valeur de jeu de (b) est égale au $\max(4-0; 0-0) = 4$.

**Q18.** Compléter le reste de l'arbre du **DR**.
Donner la case à jouer avec cette profondeur de recherche pour optimiser le gain d'Alice.

On propose la fonction **NegaAwale (jeu:dict, profondeur_max:int, profondeur: int) → int** incomplète suivante. Cette fonction applique l'algorithme *NegaMax* décrit précédemment. La fonction **max_vals** sera explicitée plus tard, c'est elle qui permet la détermination du maximum et du coup à jouer.

```
def NegaAwale (jeu, profondeur_max, profondeur):
    if                  : # Condition 1
        if (tour_joueur1(jeu) and gagnant(jeu) == jeu['joueur1']) or
           (not(tour_joueur1(jeu)) and gagnant(jeu) == jeu['joueur2']):
            return 500
        elif (tour_joueur1(jeu) and gagnant(jeu) == jeu['joueur2']) or
           (not(tour_joueur1(jeu)) and gagnant(jeu) == jeu['joueur1']):
            return -500
        else: # Egalité
            return 0
    elif                  : # Condition 2
        return 0
    else:
        choix_cases = cases_possibles(jeu)
        vals_jeu = []
        for case in choix_cases :
            # Instruction 1 : Détermination du gain et du nouveau jeu
            # Instruction 2 : Remontée de la valeur de jeu du noeud
               enfant
            vals_jeu.append([case, g-p])
return max_vals(vals_jeu, profondeur)
```

**Q19.** Compléter les lignes Condition 1, Condition 2, Instruction 1 et Instruction 2 de la fonction **NegaAwale** proposée pour réaliser l'algorithme MinMax.

La fonction **max_vals** ne doit pas se contenter de renvoyer la valeur de jeu maximale parmi les valeurs estimées. En effet, ce n'est pas la valeur de jeu maximale qu'on recherche mais l'indice de la case correspondant à celle-ci. On doit donc distinguer deux cas :

- soit le nœud père est le nœud de départ : dans ce cas, on renvoie l'indice (un entier) de la case correspondant à la valeur de jeu maximale ;
- soit le nœud père est un nœud intermédiaire : dans ce cas, on renvoie la valeur de jeu maximale (un entier).

**Q20.** Proposer une fonction **max_vals(vals_jeu:list, profondeur:int) → int** qui réalise ce qui est demandé précédemment. *On supposera pour simplifier qu'en cas de maximum multiple, c'est le premier maximum trouvé qui est conservé.*

On rappelle le programme de jeu dans le cas d'une partie entre deux joueurs humains :

```
1 def awale_jcj(nom_joueur1, nom_joueur2)
2     jeu = initialisation(nom_joueur1, nom_joueur2)
3     jeu_continue = True
4     while jeu_continue :
5         affiche( jeu['plateau'] )
6         case_choisie = int(input("Choisir une case : "))
7         jeu_continue = tour_jeu(jeu, case_choisie)
8     return gagnant(jeu)
```

**Q21.** Indiquer le numéro de la (ou des) ligne(s) qui doit (ou doivent) être modifiée(s) pour une partie entre deux joueurs IA et proposer une modification en prenant une profondeur maximale de 6.

### III.3 - Bibliothèque d'ouverture

Afin d'améliorer l'efficacité de l'IA, il est possible pour les premiers coups d'effectuer une recherche dans une base de données relationnelle. En effet, l'exploration en profondeur de l'ensemble des coups possibles est très coûteuse et on préfère s'appuyer sur l'historique des parties pour déterminer les configurations du plateau qui seront les plus avantageuses.

La base de données contient des informations sur chaque joueur, ainsi que les parties effectuées entre les joueurs.

| **Joueur** | | | | |
|---|---|---|---|---|
| **id_Joueur** | nom | prenom | niveau | naissance |
| 18571 | Martin | Jean | 2048 | 23/02/1958 |
| 18572 | Dupond | Marie | 2103 | 03/01/1972 |
| 18573 | Develion | Théo | 1857 | 05/10/2004 |

| **Partie** | | | | | |
|---|---|---|---|---|---|
| **id_Partie** | id_joueur1 | id_joueur2 | resultat | jour | jeu |
| 1 | 1547 | 1568 | 0.5 | 08/01/2001 | 'egai...' |
| 2 | 1204 | 3 | 0 | 12/07/1998 | 'egaj...' |
| 3 | 4 | 2 | 1 | 15/07/2018 | 'egbi...' |

La table Joueur contient les attributs suivants :

- **id_Joueur** : identifiant d'un joueur (entier, clé primaire) ;
- nom : nom du joueur (chaîne de caractères) ;
- prenom : prénom du joueur (chaîne de caractères) ;
- niveau : niveau maximal atteint par le joueur au cours de sa carrière (entier) ;
- naissance : date de naissance du joueur (date).

La table Partie contient les attributs suivants :

- **id_Partie** : identifiant de la partie (entier, clé primaire) ;
- id_joueur1 : identifiant du joueur débutant la partie (entier) ;
- id_joueur2 : identifiant du second joueur (entier) ;
- resultat : 1 est une victoire du joueur1, 0.5 une égalité et 0 une victoire du joueur2 (flottant) ;
- jour : date du jour de la partie (date) ;
- jeu : liste des coups successifs de la partie, sans inversion du plateau, stockée sous forme d'une chaîne de caractères. 'egai...' signifie que le joueur1 a joué la 5$^{e}$ case (d'indice 4) représentée par la lettre 'e', puis le joueur2 a joué la 7$^{e}$ case (d'indice 6) représentée par la lettre 'g', puis le joueur1 a joué la 1$^{re}$ case et le joueur2 a répondu par la 9$^{e}$ case et ainsi de suite.

*Remarque : l'opérateur LIKE est utilisé pour comparer des chaînes de caractères dans la clause WHERE des requêtes SQL. Ce mot-clé permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher les enregistrements dont la valeur d'une colonne commence par telle ou telle lettre. Le caractère _ (underscore) représente n'importe quel caractère, mais un seul caractère uniquement alors que le caractère pourcentage % peut être remplacé par un nombre quelconque (et possiblement nul) de caractères. Par exemple parmi une recherche dans les communes de France, nom LIKE '_ff%f%' ne renvoie que Offendorf alors que remplacer le _ par un % renvoie Pfaffenhoffen et Staffelfelden en plus de Offendorf.*

**Q22.** Écrire une requête SQL permettant d'extraire les identifiants des joueurs ayant un niveau strictement supérieur au score 1900.

**Q23.** Écrire une requête SQL permettant de déterminer le pourcentage de victoires du joueur1 pour les parties où la case d'indice 0 a été jouée en premier.

**Q24.** Écrire une requête SQL permettant d'afficher le nom et le prénom des 3 joueurs ayant le niveau le plus élevé.

**Q25.** Écrire une requête SQL permettant de déterminer les joueurs ayant plus de cent victoires lorsqu'ils commencent la partie. La requête doit renvoyer le nom, le prénom et le nombre de victoires de ces joueurs classés par ordre décroissant du nombre de victoires.

# ANNEXE

## Rappels des syntaxes en Python

Seules les commandes listées ici peuvent être utilisées. Il n'est pas autorisé de faire appel à certaines fonctions Python déjà implémentées (min, max, sort, ...).

| Fonctionnalités | Commandes Python |
|---|---|
| définir une liste | L = [1,2,3] |
| définir un dictionnaire | dic = {'a':0,'b':1,'c':2} |
| accéder à un élément | L[0] renvoie 1<br>dic['a'] renvoie 0 |
| extraire une sous-liste | L[1:2] renvoie [2] |
| vérifier si une clé est dans un dictionnaire | 'a' **in** dic renvoie True |
| ajouter un élément à une liste | L.append(5) |
| supprimer le dernier élément d'une liste et le renvoyer | a = L.pop() |
| copier une liste | L2 = L.copy() |
| ajouter un élément à un dictionnaire | dic['d'] = 4 |
| définir une chaîne de caractères | mot = 'Python' |
| taille d'une chaîne, d'une liste ou d'un dictionnaire | len(mot) |
| extraire des caractères | mot[2:6] |
| concaténer des chaînes ou des listes | 'cc'+'inp' donne 'ccinp' |
| dupliquer des chaînes ou des listes | 'c'*3 donne 'ccc' |
| convertir en flottant, en entier, en chaîne, en liste | float(s), int(s), str(L), list(s) |
| définir une chaîne de caractères contenant une tabulation | chaine = 'a\t b'<br>>>> print(chaine)<br>a  b |
| parcours en valeur d'un dictionnaire | **for** v **in** dic.values(): print(v) |
| parcours des clés d'un dictionnaire | **for** c **in** dic : print(c) |
| parcours des clés et des valeurs d'un dictionnaire | **for** c, v **in** dic.items() : print(c, v) |

**FIN**



IMPRIMERIE NATIONALE – 24 1107 – D'après documents fournis

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom : _____

Prénom : _____

Emplacement QR Code

Filière : PSI

Session : 2024

Épreuve de : INFORMATIQUE

**Consignes**

- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

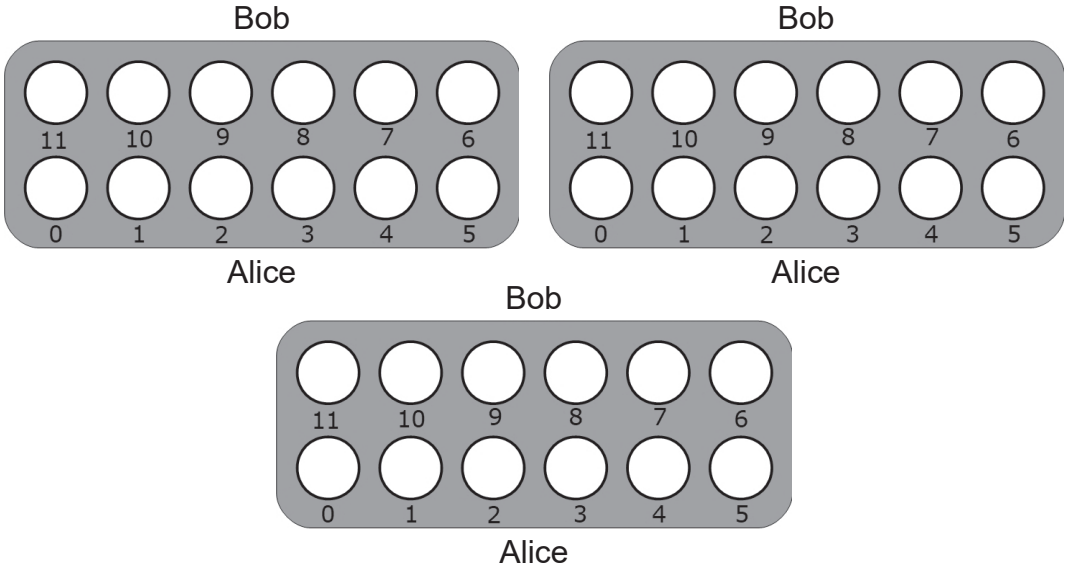


# DOCUMENT RÉPONSE

**Ce Document Réponse doit être rendu dans son intégralité.**

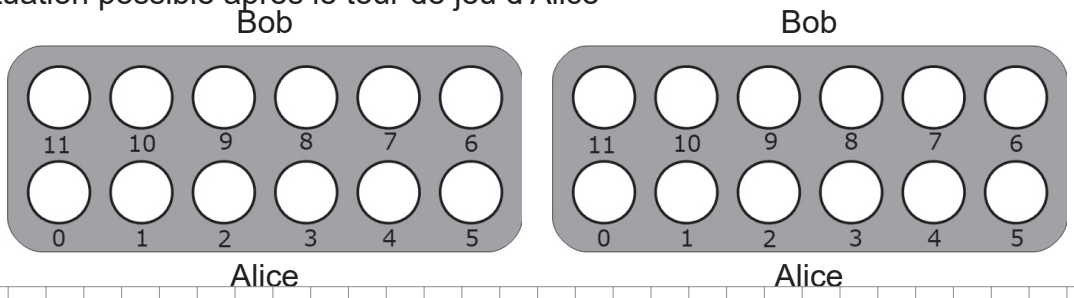
**Q1** - Indice et justification

**Q2** - Situation possible après le tour d'Alice



**NE RIEN ÉCRIRE DANS CE CADRE**

**Q3** - Situation possible après le tour de jeu d'Alice




**Q4** - Parité de **jeu['n']**. Fonction **tour_joueur1(jeu:dict) → bool**

**Q5** - Fonction **tourner_plateau(jeu:dict) → None**

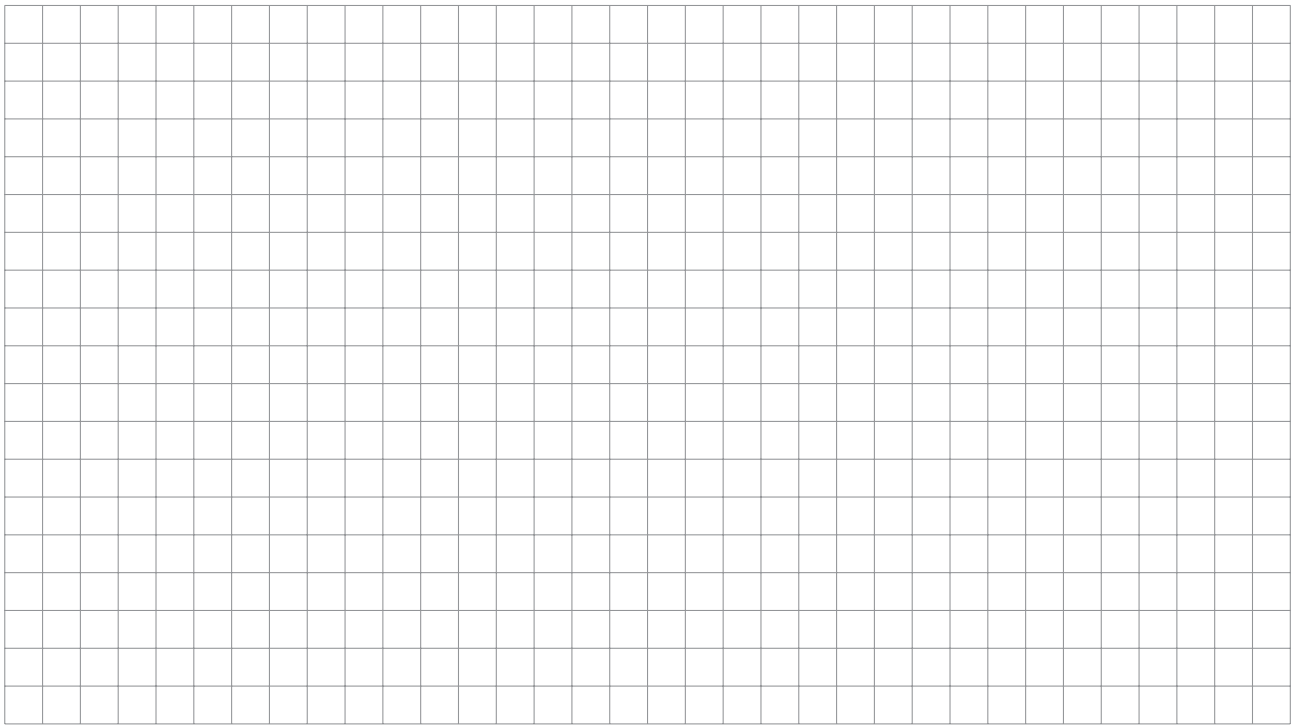
**Q6 -** Maximum de graines. Bits nécessaires

**Q7 -** Fonction **copie(jeu:dict) → dict**

**Q8 -** Fonction **deplacer_graines(plateau:[int], case:int) → int**



**Q9 -** Fonction **case_ramassable(plateau:[int], case:int) → bool**



CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : PSI

Session : 2024

Épreuve de : INFORMATIQUE

**Consignes**

- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*



**Q10** - Fonction récursive **ramasser_graines(plateau:[int], case:int) → int**

**NE RIEN ÉCRIRE DANS CE CADRE**

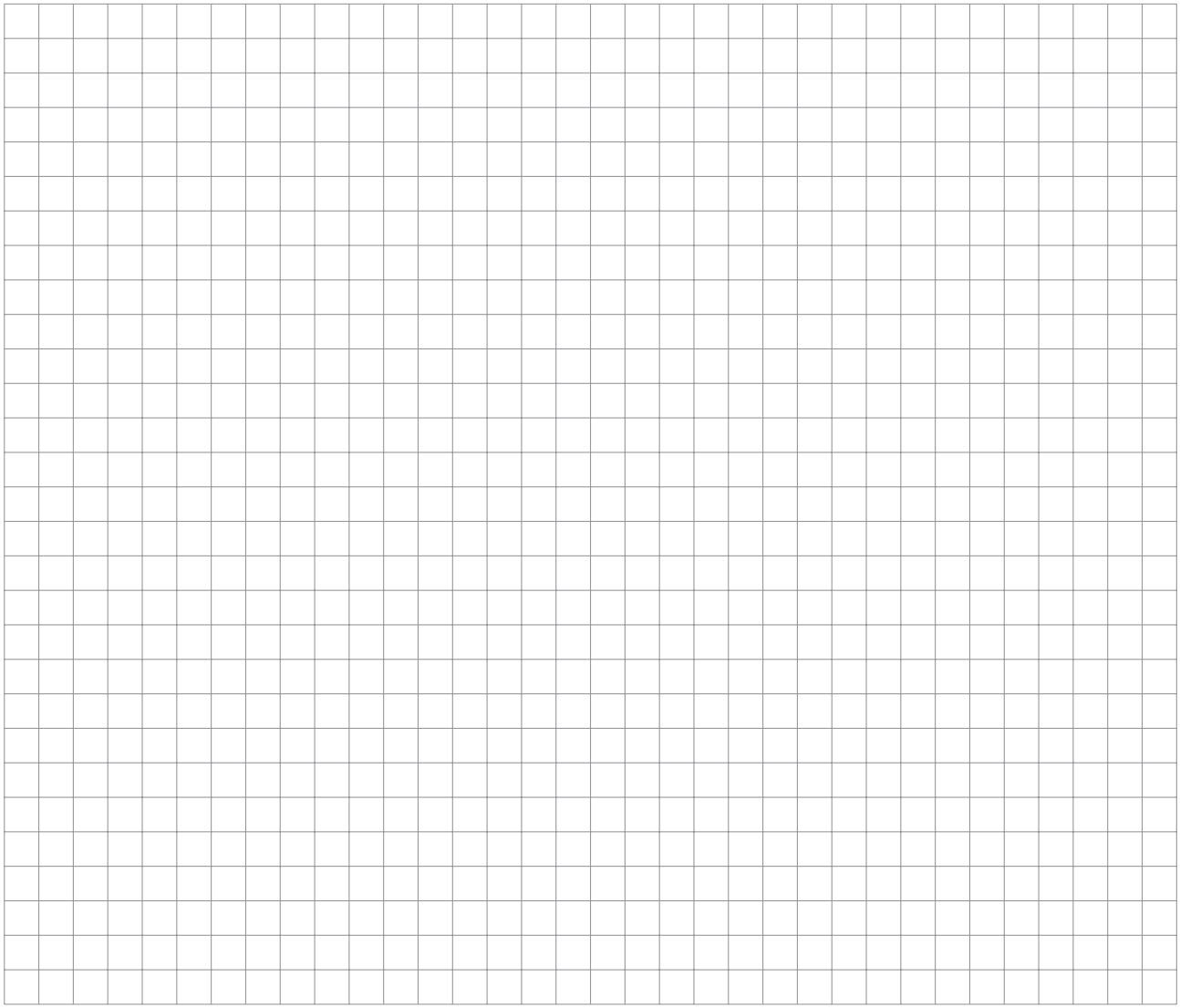
**Q11** - Fonction **test_famine(plateau:[int], case:int) → bool**

**Q12** - Expression de la condition test

**Q13 -** Fonction **cases_possibles(jeu:dict) → [int]**



**Q14 -** Fonction **tour_suivant(jeu:dict) → bool**



**Q15** - Instruction 1, instruction 2, condition 1 et instruction 3

Instruction 1

Instruction 2

Condition 1

Instruction 3

**Q16** - Fonction **gagnant(jeu:dict) → str**

CONCOURS COMMUN

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : PSI

Session : 2024

Épreuve de : INFORMATIQUE

**Consignes**

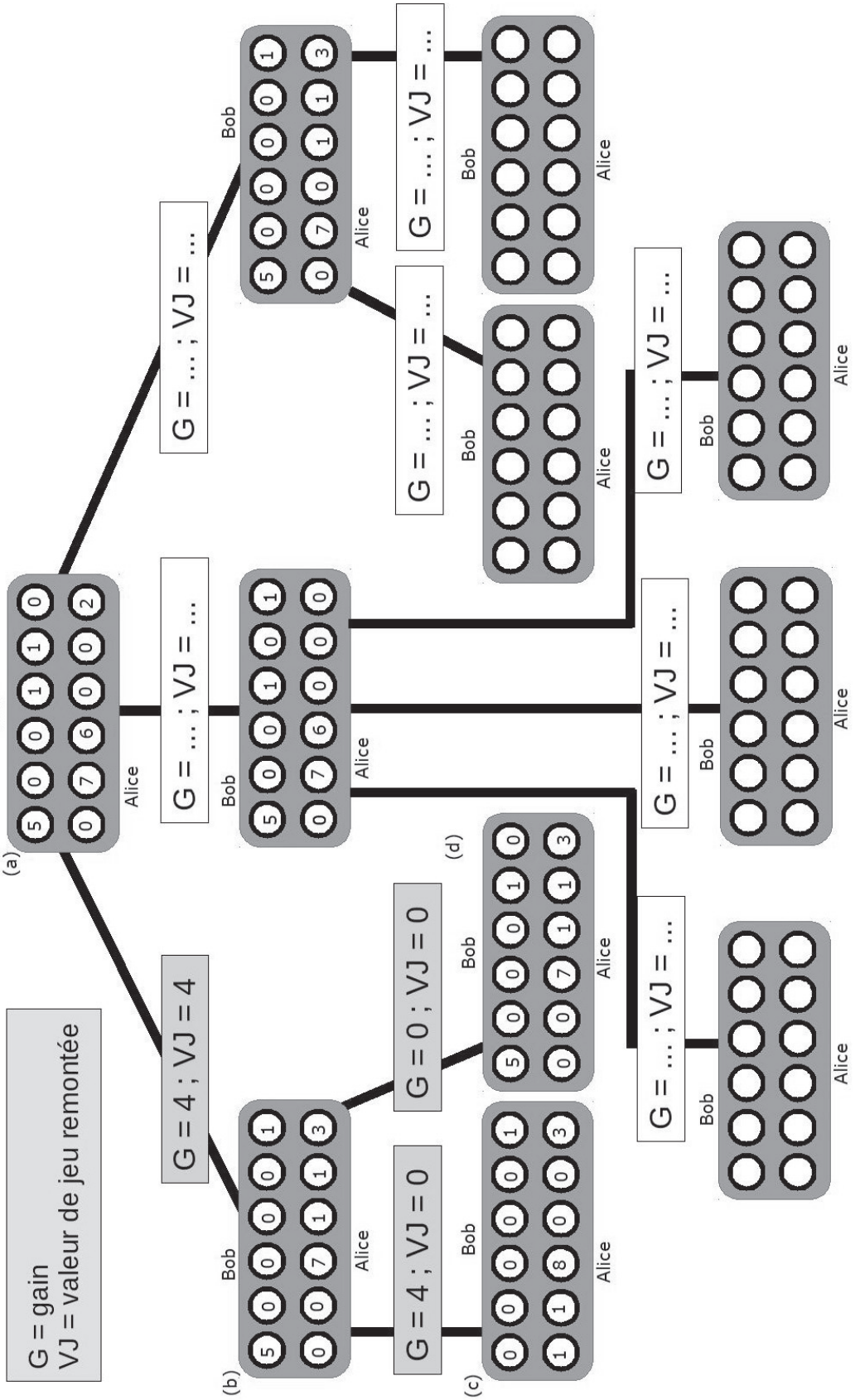
- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*



**Q17** - Fonction **gain(jeu:dict, case:int) → int,dict**

**NE RIEN ÉCRIRE DANS CE CADRE**

**Q18** - Arbre des jeux possibles à compléter



**Q18** - Numéro de la case à jouer avec cette profondeur de recherche

**Q19** - Compléter NegaAwale

Condition 1

Condition 2

Instruction 1

Instruction 2

**Q20** - Fonction **max_vals(vals_jeu:list, profondeur:int) → int**

**Q21** - Numéro(s) de(s) ligne(s) à modifier et modification(s) à apporter

**Q22** - Identifiants des joueurs ayant un niveau strictement supérieur au score 1900

**Q23** - Pourcentage de victoires en commençant la partie par la première case

**Q24** - Nom et prénom des 3 joueurs ayant le niveau le plus élevé

**Q25** - Les joueurs ayant plus de cent victoires en commençant la partie

**FIN**

# Travailler les chapitres de ce sujet

Prepa Help suit le programme de PSI chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

## Chapitres liés à ce sujet

- Bases de Python : expressions, types, instructions, fonctions
- Récursivité
- Architecture de l'ordinateur et représentation des nombres

## Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

**prepahelp.com/#essai**

Une question sur le concours ou sur ta filière : **écrire sur WhatsApp**

Les autres années et épreuves : prepahelp.com/annales/e3a/psi