# Correction CB – CCINP PC 2024 – Le jeu de l'awalé

## Partie I - Présentation et règles

1. **Cases qu'Alice peut choisir de jouer :**
   On rappelle la situation (figure 4) :

| Camp de Bob (adversaire) | | | | | |
|---|---|---|---|---|---|
| case 11 | case 10 | case 9 | case 8 | case 7 | case 6 |
| 1 | 0 | 1 | 0 | 1 | 2 |
| Camp d'Alice (joueur actif) | | | | | |
| case 0 | case 1 | case 2 | case 3 | case 4 | case 5 |
| 0 | 0 | 2 | 0 | 16 | 5 |

   Les cases du camp d'Alice sont les cases 0 à 5. Pour être jouable, une case doit être non vide et respecter la règle de la famine (ne pas laisser le camp adverse vide après récolte).

   - Case 0 : 0 graine → **non jouable** (vide).
   - Case 1 : 0 graine → **non jouable** (vide).
   - Case 2 : 2 graines → on sème dans les cases 3 et 4. La dernière graine tombe en case 4 (camp d'Alice), pas de récolte possible côté adverse. Le camp adverse n'est pas vidé → **jouable**.
   - Case 3 : 0 graine → **non jouable** (vide).
   - Case 4 : 16 graines → on sème dans 16 cases successives (en sautant la case 4 d'origine) : cases 5, 6, 7, 8, 9, 10, 11, 0, 1, 2, 3, puis 5, 6, 7, 8, 9. La dernière graine tombe en **case 9** (camp de Bob). Après semence et éventuelle récolte, le camp adverse n'est pas vidé → **jouable**.
   - Case 5 : 5 graines → on sème dans les cases 6, 7, 8, 9, 10. La dernière graine tombe en case 10. On vérifie la récolte : case 10 contient alors $0 + 1 = 1$ graine (pas 2 ou 3), pas de récolte. Le camp adverse n'est pas vidé → **jouable**.

   Les cases jouables sont donc : $\boxed{2, 4 \text{ et } 5}$.

2. **Situation après le coup d'Alice pour chaque choix :**

   ▷ **Alice joue la case 2** (2 graines) :

   - Semence : case 2 → 0, case 3 : $0 + 1 = 1$, case 4 : $16 + 1 = 17$.
   - Dernière graine en case 4 (camp d'Alice) → pas de récolte (condition 1 non vérifiée : la case n'est pas dans le camp adverse).

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 2 |
| 0 | 0 | 0 | 1 | 17 | 5 |

   Gain : $\boxed{0 \text{ graine}}$

   ▷ **Alice joue la case 4** (16 graines) :

   - Semence : on prend les 16 graines de la case 4 et on les sème une par une dans le sens direct : cases 5, 6, 7, 8, 9, 10, 11, 0, 1, 2, 3 (11 graines), on saute la case 4 (case d'origine), puis cases 5, 6, 7, 8, 9 (5 graines restantes). La dernière graine tombe en **case 9**.
   - Après semence :

| case 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 1 | 0 | 7 | 4 | 3 | 2 | 3 | 1 | 2 |

   - Récolte depuis la case 9 (camp de Bob) :
     - Case 9 : 3 graines → récoltable.
     - Case 8 : 2 graines → récoltable.
     - Case 7 : 3 graines → récoltable.
     - Case 6 : 4 graines → **arrêt** (ni 2 ni 3 graines).
   - On vérifie que la récolte ne vide pas le camp adverse : il reste les cases 6 (4), 10 (1), 11 (2), soit 7 graines → pas de famine.

- Gain : $3 + 2 + 3 = 8$ graines récoltées.

| | | | | | |
|---|---|---|---|---|---|
| 2 | 1 | 0 | 0 | 0 | 4 |
| 1 | 1 | 3 | 1 | 0 | 7 |

Gain : $\boxed{8 \text{ graines}}$

▷ **Alice joue la case 5** (5 graines) :

- Semence : case 5 → 0, case 6 : $2 + 1 = 3$, case 7 : $1 + 1 = 2$, case 8 : $0 + 1 = 1$, case 9 : $1 + 1 = 2$, case 10 : $0 + 1 = 1$.
- Dernière graine en case 10 (camp de Bob). Case 10 contient 1 graine → pas de récolte (condition 2 non vérifiée).

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 1 | 2 | 3 |
| 0 | 0 | 2 | 0 | 16 | 0 |

Gain : $\boxed{0 \text{ graine}}$

3. **Figure 5, situations après le tour d'Alice :**

▷ **Situation de gauche :**

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 5 |

Alice ne peut jouer que la case 5 (seule case non vide de son camp). La case 5 contient 5 graines, semées en cases 6, 7, 8, 9, 10. Après semence :

| | | | | | |
|---|---|---|---|---|---|
| 0 | 2 | 2 | 2 | 2 | 2 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Dernière graine en case 10. Case 10 : 2 graines → récoltable. Case 9 : 2 graines → récoltable. Case 8 : 2 graines → récoltable. Case 7 : 2 graines → récoltable. Case 6 : 2 graines → récoltable.

Mais si on récolte toutes ces cases, le camp adverse serait entièrement vidé (il ne resterait que la case 11 avec 0 graine). Cela viole la **règle de la famine** : la récolte est donc **annulée**.

Le plateau reste :

| | | | | | |
|---|---|---|---|---|---|
| 0 | 2 | 2 | 2 | 2 | 2 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Gain : $\boxed{0 \text{ graine}}$

▷ **Situation de droite :**

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 5 |

Alice ne peut jouer que la case 5 (seule case non vide). Semence de 5 graines en cases 6, 7, 8, 9, 10 :

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 2 | 2 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Dernière graine en case 10. Case 10 : 2 graines → récoltable. Case 9 : 2 graines → récoltable. Case 8 : 2 graines → récoltable. Case 7 : 2 graines → récoltable. Case 6 : 2 graines → récoltable.

Si on récolte ces 5 cases, il reste la case 11 avec 1 graine dans le camp adverse : le camp adverse n'est pas vidé. La récolte est **valide**.

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Gain : $\boxed{2 \times 5 = 10 \text{ graines}}$

## Partie II - Programmation de la structure de jeu

### II.1 - Représentation du jeu

4. Lorsque c'est au tour du joueur1 de jouer, `jeu['n']` est **pair** (le joueur1 commence avec $n = 0$, puis joue à $n = 2, 4, \ldots$).

```python
def tour_joueur1(jeu):
    return jeu['n'] % 2 == 0
```

5. Fonction `tourner_plateau` :

```python
def tourner_plateau(jeu):
    plateau = jeu['plateau']
    jeu['plateau'] = plateau[6:12] + plateau[0:6]
```

**Remarque :** On utilise le slicing pour échanger les deux moitiés du plateau. Les cases 6 à 11 deviennent les cases 0 à 5 et inversement.

On peut ne pas utiliser les sous-listes du tout. Une solution un peu à la main est :

```python
def tourner_plateau(jeu):
    plateau = []
    for i in range(6,12):
        plateau.append(jeu['plateau'][i])
    for i in range(0,6):
        plateau.append(jeu['plateau'][i])
    jeu['plateau'] = plateau
```

6. **Maximum de graines dans une case :**

Au début, il y a $12 \times 4 = 48$ graines au total. Une case peut potentiellement contenir toutes les graines (par accumulation au cours du jeu), soit au maximum $\boxed{48 \text{ graines}}$.

**Nombre de bits nécessaires :** Il faut coder les entiers de 0 à 48. On a $2^5 = 32 < 48 < 64 = 2^6$, donc il faut $\boxed{6 \text{ bits}}$.

7. Fonction `copie` :

```python
def copie(jeu):
    nouveau = {}
    nouveau['joueur1'] = jeu['joueur1']
    nouveau['joueur2'] = jeu['joueur2']
    nouveau['score'] = jeu['score'].copy()
    nouveau['n'] = jeu['n']
    nouveau['plateau'] = jeu['plateau'].copy()
    return nouveau
```

**Remarque :** Il est indispensable de faire un `.copy()` sur les listes (`score` et `plateau`) pour obtenir une copie profonde. Les chaînes de caractères et les entiers étant immuables, une simple affectation suffit.

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

8. Fonction `deplacer_graines` :

```python
def deplacer_graines(plateau, case):
    nb = plateau[case]        # On prend toutes les graines
    plateau[case] = 0         # La case d'origine est vidée
    pos = case                # Position courante
    for i in range(nb):
        pos = (pos + 1) % 12
        if pos == case:       # On saute la case d'origine
            pos = (pos + 1) % 12
        plateau[pos] += 1
    return pos                # Indice de la dernière case semée
```

**Remarque :** Le modulo 12 assure la circularité du plateau. Le test `pos == case` gère le cas où la case d'origine contient plus de 11 graines.

9. Fonction `case_ramassable` :

```
def case_ramassable(plateau, case):
    # La case est dans le camp adverse (indices 6 à 11)
    # et contient exactement 2 ou 3 graines
    return case >= 6 and (plateau[case] == 2 or plateau[case] == 3)
```

10. Fonction récursive `ramasser_graines` :

```
def ramasser_graines(plateau, case):
    if not case_ramassable(plateau, case):
        return 0
    # On ramasse les graines de cette case
    nb = plateau[case]
    plateau[case] = 0
    # On continue la récolte dans le sens inverse (case précédente)
    return nb + ramasser_graines(plateau, case - 1)
```

**Remarque :** La récolte se fait dans le sens inverse de la semence. Si la case précédente ne vérifie pas les conditions, la récursion s'arrête et on renvoie le nombre total de graines récoltées.

11. Fonction `test_famine` :

```
def test_famine(plateau, case):
    # On simule le coup sur une copie du plateau
    copie_plateau = plateau.copy()
    derniere = deplacer_graines(copie_plateau, case)
    ramasser_graines(copie_plateau, derniere)
    # On vérifie qu'il reste des graines dans le camp adverse
    for i in range(6, 12):
        if copie_plateau[i] > 0:
            return True  # La condition 3 est vérifiée
    return False          # Camp adverse vide : famine
```

**Remarque :** On travaille sur une copie du plateau pour ne pas modifier l'original. Après simulation du coup complet (semence + récolte), on vérifie qu'il reste au moins une graine dans le camp adverse.

12. **Compléter la condition `test` :**
On recopie la fonction complète ; la ligne complétée est signalée par ← :

```
def test_case(plateau, case):
        """Vérifie si la case choisie par le joueur est acceptable
        renvoie True si la case est acceptable, False sinon"""
        condition3 = test_famine(plateau, case)
        # Case acceptable
        test = (0 <= case <= 5) and (plateau[case] > 0) and condition3   ←
        return test
```

La condition 1 (`0 <= case <= 5`) vérifie que la case est dans le camp du joueur actif, la condition 2 (`plateau[case] > 0`) vérifie que la case n'est pas vide, et `condition3` vérifie l'absence de famine.

13. Fonction `cases_possibles` :

```
def cases_possibles(jeu):
    resultat = []
    plateau = jeu['plateau']
    for case in range(6):
        if test_case(plateau, case):
            resultat.append(case)
    return resultat
```

**Remarque :** On parcourt les 6 cases du joueur actif et on teste chacune d'elles avec `test_case`. La fonction `test_case` utilise `test_famine` qui travaille sur une copie, donc le plateau n'est pas modifié.

14. Fonction `tour_suivant` :

```
def tour_suivant(jeu):
    # Un joueur a au moins 25 graines
    if jeu['score'][0] >= 25 or jeu['score'][1] >= 25:
        return False
    # Plus de 100 tours joués
    if jeu['n'] >= 100:
        return False
    # 3 graines ou moins sur le plateau
    total = 0
    for i in range(12):
        total += jeu['plateau'][i]
    if total <= 3:
        return False
    # Le joueur actif n'a plus de case jouable
    if cases_possibles(jeu) == []:
        return False
    return True
```

15. **Compléter la fonction `tour_jeu` :**
On recopie la fonction complète :

```
def tour_jeu(jeu, case):
    plateau = jeu['plateau']
    if test_case(plateau, case):  # La case jouée est acceptable
        derniere = deplacer_graines(plateau, case) Instruction 1
        graines_gagnees = ramasser_graines(plateau, derniere) Instruction 2
        """Pour augmenter le score, il faut savoir qui joue grâce à la
           parité du nombre de tours"""
        if tour_joueur1(jeu): Condition 1
            jeu['score'][0] = jeu['score'][0] + graines_gagnees
        else:
            jeu['score'][1] = jeu['score'][1] + graines_gagnees
        jeu['n'] = jeu['n'] + 1 Instruction 3
        tourner_plateau(jeu)  # Echanger les plateaux
        return tour_suivant(jeu)
    else:
        print("La case choisie n'est pas valable")
        return True
```

16. Fonction `gagnant` :

```
def gagnant(jeu):
    # Ramasser les graines restantes de chaque camp
    # Les cases 0-5 sont celles du joueur actif
    plateau = jeu['plateau']
    graines_actif = 0
    graines_adversaire = 0
    for i in range(6):
        graines_actif += plateau[i]
    for i in range(6, 12):
        graines_adversaire += plateau[i]
    # Attribuer au bon joueur
    if tour_joueur1(jeu):
```

```
        jeu['score'][0] += graines_actif
        jeu['score'][1] += graines_adversaire
    else:
        jeu['score'][1] += graines_actif
        jeu['score'][0] += graines_adversaire
    # Comparer les scores
    if jeu['score'][0] > jeu['score'][1]:
        return jeu['joueur1']
    elif jeu['score'][1] > jeu['score'][0]:
        return jeu['joueur2']
    else:
        return "égalité"
```

**Remarque :** Le tableau `jeu['score']` associe toujours l'indice 0 au joueur1 et l'indice 1 au joueur2, quelle que soit la parité du tour. En revanche, comme le plateau est inversé à chaque tour, les cases 0 à 5 correspondent au joueur actif (joueur1 si `jeu['n']` est pair, joueur2 sinon). Il faut donc vérifier à qui appartiennent ces cases avant de créditer les graines restantes.

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

### III.1 - Arbre des configurations

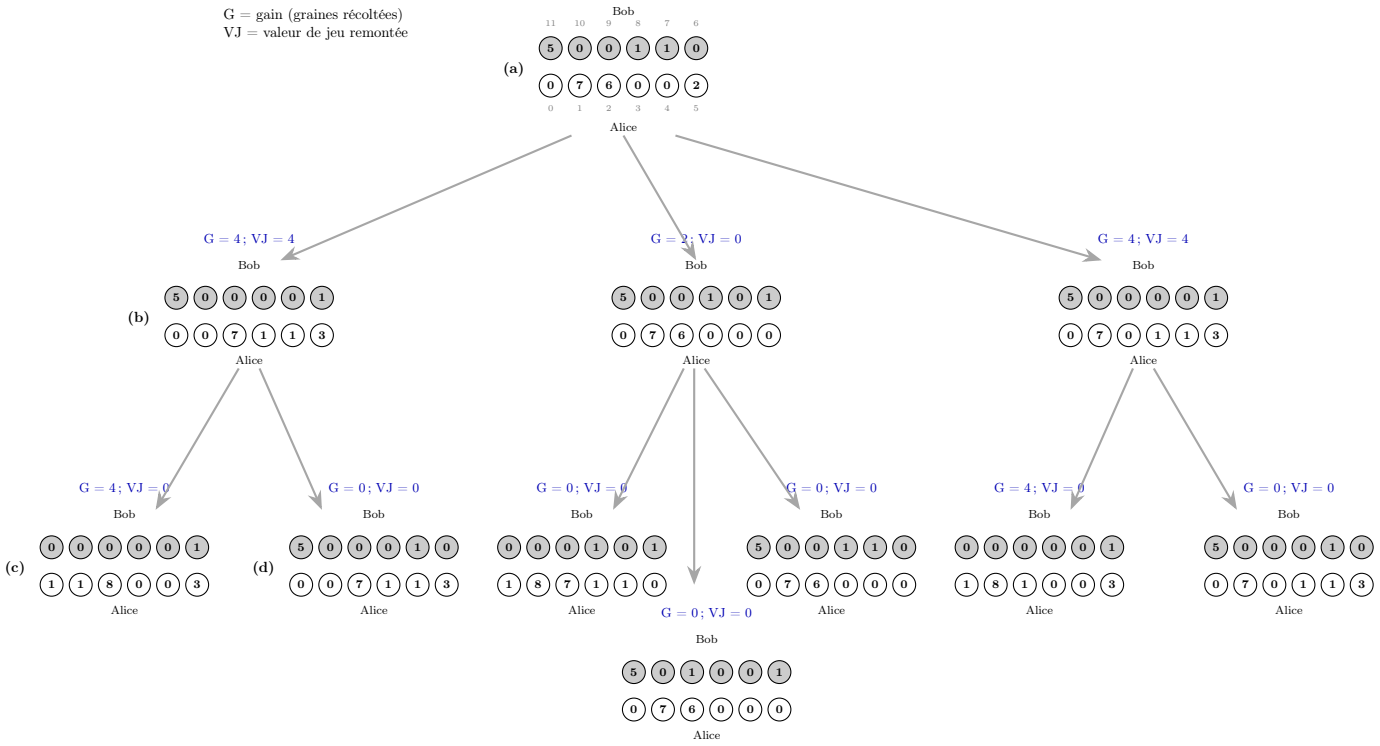
17. Fonction `gain` :

```
def gain(jeu, case):
    nouveau_jeu = copie(jeu)
    plateau = nouveau_jeu['plateau']
    # Déplacer les graines
    derniere = deplacer_graines(plateau, case)
    # Ramasser les graines
    graines_gagnees = ramasser_graines(plateau, derniere)
    # Mettre à jour le score
    if tour_joueur1(nouveau_jeu):
        nouveau_jeu['score'][0] += graines_gagnees
    else:
        nouveau_jeu['score'][1] += graines_gagnees
    # Incrémenter le nombre de tours
    nouveau_jeu['n'] += 1
    # Échanger les plateaux
    tourner_plateau(nouveau_jeu)
    return graines_gagnees, nouveau_jeu
```

**Remarque :** On travaille sur une copie profonde du jeu pour ne pas modifier les arguments d'entrée. La fonction reproduit les étapes de `tour_jeu` mais sans vérification de la validité (supposée acquise) et en renvoyant le gain et le nouvel état.

### III.2 - Algorithme MinMax

18. **Arbre des jeux possibles :**




**Case à jouer :** Alice doit jouer la **case 5** (branche centrale) car G-VJ est maximal

19. **Compléter `NegaAwale` :**

On recopie la fonction complète :

```
1  def NegaAwale(jeu, profondeur_max, profondeur):
       if not tour_suivant(jeu):                                  Condition 1
3          if (tour_joueur1(jeu) and
               gagnant(jeu) == jeu['joueur1']) or \
5             (not(tour_joueur1(jeu)) and
               gagnant(jeu) == jeu['joueur2']):
               return 500
           elif (tour_joueur1(jeu) and
                 gagnant(jeu) == jeu['joueur2']) or \
10               (not(tour_joueur1(jeu)) and
                  gagnant(jeu) == jeu['joueur1']):
               return -500
           else:  # Egalité
               return 0
15     elif profondeur >= profondeur_max:                         Condition 2
16         return 0
       else:
           choix_cases = cases_possibles(jeu)
           vals_jeu = []
20         for case in choix_cases:
               g, nouveau_jeu = gain(jeu, case)                   Instruction 1
22             p = NegaAwale(nouveau_jeu,
                             profondeur_max, profondeur + 1)      Instruction 2
24             vals_jeu.append([case, g - p])
25         return max_vals(vals_jeu, profondeur)
```

**Explications :**

- **Condition 1** (`not tour_suivant(jeu)`) : on teste si le nœud est une feuille, c'est-à-dire si le jeu est terminé.
- **Condition 2** (`profondeur >= profondeur_max`) : on a atteint la profondeur maximale d'exploration, on renvoie une valeur de jeu nulle.

- **Instruction 1** (`g, nouveau_jeu = gain(jeu, case)`) : on calcule le gain et le nouvel état du jeu pour le coup considéré.
- **Instruction 2** (`p = NegaAwale(...)`) : on calcule récursivement la valeur de jeu du nœud enfant. La profondeur est incrémentée de 1.

**Remarque :** la rédaction de `NegaAwale` est fort étrange puisqu'on ne connaît pas ce que fait `max_vals`. Mais bon, il faut faire avec (on peut éventuellement faire au brouillon la question suivante d'abord).

20. Fonction `max_vals` :

```
1  def max_vals(vals_jeu, profondeur):
       # vals_jeu est une liste de [case, valeur]
       meilleur_indice = 0
       meilleur_valeur = vals_jeu[0][1]
5      for i in range(1, len(vals_jeu)):
           if vals_jeu[i][1] > meilleur_valeur:
               meilleur_valeur = vals_jeu[i][1]
               meilleur_indice = i
       if profondeur == 0:
10         # Nœud de départ : on renvoie l'indice de la case
           return vals_jeu[meilleur_indice][0]
       else:
           # Nœud intermédiaire : on renvoie la valeur de jeu
           return meilleur_valeur
```

**Remarque :** L'inégalité stricte > garantit qu'en cas de maximum multiple, c'est le premier rencontré qui est conservé (conformément à l'énoncé).

**Remarque :** La fonction ne renvoit pas toujours le même type d'objet. C'est... non-conventionnel. Et il vaudrait mieux l'éviter dans le cas général. Mais ici, on n'avait pas le choix puisque le sujet le demandait.

La raison sous-jacente ici est que pour la profondeur minimale, on veut savoir quel coup jouer. Mais en dessous, on a juste besoin de savoir le score des configurations.

Si vous étiez amenés à écrire vous-mêmes une telle fonction, renvoyez juste les deux à chaque niveau.

21. **Modification pour deux joueurs IA :**

Il faut modifier la **ligne 6**. On recopie la fonction complète :

```
1  def awale_jcj(nom_joueur1, nom_joueur2):
       jeu = initialisation(nom_joueur1, nom_joueur2)
       jeu_continue = True
       while jeu_continue:
5          affiche(jeu['plateau'])
           case_choisie = NegaAwale(jeu, 6, 0)   ← ligne 6 modifiée
7          jeu_continue = tour_jeu(jeu, case_choisie)
       return gagnant(jeu)
```

**Remarque :** sans l'instruction `input`, les coups vont s'enchaîner sans pause. Il conviendrait peut-être de permettre à l'utilisateur de faire apparaître un coup à la fois à chaque fois.

### III.3 - Bibliothèque d'ouverture (SQL)

22. Identifiants des joueurs de niveau strictement supérieur à 1900 :

```
1  SELECT id_Joueur
   FROM Joueur
   WHERE niveau > 1900;
```

23. Pourcentage de victoires du joueur1 pour les parties commençant par la case d'indice 0 :

La case d'indice 0 correspond à la lettre `'a'` (la 1re lettre). On cherche les parties dont le champ `jeu` commence par `'a'`, et on calcule le rapport entre le nombre de victoires strictes (`resultat = 1`) et le nombre total de telles parties.

```sql
SELECT 100.0 *
  (SELECT COUNT(*) FROM Partie
   WHERE jeu LIKE 'a%' AND resultat = 1)
  / COUNT(*) AS pourcentage
FROM Partie
WHERE jeu LIKE 'a%';
```

**Remarque :** La requête principale compte le nombre total de parties commençant par la case 0, tandis que la sous-requête compte uniquement les victoires strictes du joueur1 (`resultat = 1`, excluant les égalités à 0.5).

**Remarque :** On pourrait utiliser `SUM` pour compter les victoires, mais cela pose un problème avec les cas d'égalités. La solution précédente me paraît la meilleure.

**Remarque :** On peut éviter la sous-requête en utilisant un polynôme de Lagrange qui vaut 0 pour 0, 0 pour 0.5 et 1 pour 1. Cela donne :

```sql
SELECT 100.0 *
        SUM(resultat*(resultat-0.5)*2)/COUNT(*) AS pourcentage
FROM Partie
WHERE jeu LIKE 'a%';
```

24. Nom et prénom des 3 joueurs ayant le niveau le plus élevé :

```sql
SELECT nom, prenom
FROM Joueur
ORDER BY niveau DESC
LIMIT 3;
```

25. Joueurs ayant plus de 100 victoires en commençant la partie :

```sql
SELECT nom, prenom, COUNT(*) AS nb_victoires
FROM Joueur AS J
JOIN Partie AS P ON J.id_Joueur = P.id_joueur1
WHERE P.resultat = 1
GROUP BY J.id_Joueur
HAVING nb_victoires > 100
ORDER BY nb_victoires DESC;
```

**Remarque :** On joint la table `Joueur` avec `Partie` sur la condition que le joueur est le joueur1 (celui qui commence). On ne garde que les victoires (`resultat = 1`). Le `GROUP BY` permet de compter les victoires par joueur, et le `HAVING` filtre ceux qui en ont plus de 100.