L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation sera **obligatoirement** `Python`.

# Tetris Couleurs

On souhaite programmer un jeu de puzzle dont le principe ressemble au jeu de Tetris : des blocs de couleur apparaissent à l'écran en haut d'une aire de jeu et descendent sous l'effet de la « gravité » jusqu'à reposer sur un bloc déjà présent, ou sur le bas de l'aire de jeu. Les nouveaux blocs de couleur apparaissent par groupe de $k$ blocs superposés en une tour verticale appelée « *barreau* ».

Lors de la descente, le joueur peut déplacer le barreau selon certaines règles. Il peut :

- déplacer le barreau vers la gauche ou vers la droite,
- permuter l'ordre des blocs dans le barreau,
- faire descendre le barreau « rapidement ».

Le but du jeu est de réaliser le plus grand nombre possible d'alignements d'au moins trois blocs de la même couleur. Chaque alignement de trois blocs de la même couleur (horizontalement, verticalement ou en diagonale) donne un point au joueur. Les blocs appartenant à des alignements sont retirés de l'aire de jeu, et les blocs restants sont tassés (ils descendent sous l'effet de la gravité). Les blocs ainsi tassés peuvent à nouveau former des alignements unicolores qui sont à leur tour retirés, et ainsi de suite jusqu'à ce qu'il n'y ait plus aucun alignement unicolore dans l'aire de jeu.

La partie se termine quand l'aire de jeu est trop remplie pour accueillir un nouveau barreau. Le score du joueur est alors la somme des points accumulés lors de la partie.

**Complexité.** La complexité, ou le temps d'exécution, d'un programme $P$ (fonction ou procédure) est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc.) nécessaires à l'exécution de $P$. Lorsque cette complexité dépend de deux paramètres $n$ et $m$, on dira que $P$ a une complexité en $\mathcal{O}(\varphi(n,m))$ lorsqu'il existe trois constantes $A$, $n_0$ et $m_0$ telles que la complexité de $P$ est inférieure ou égale à $A \cdot \varphi(n,m)$, pour tout $n \geq n_0$ et $m \geq m_0$.

Lorsqu'il est demandé de donner la complexité d'un programme, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

**Python.** Dans ce sujet, nous adopterons la syntaxe du langage `Python`. On rappelle qu'en `Python`, les listes sont des tableaux dynamiques à une dimension. Sur les listes, on dispose des opérations suivantes, qui ont toutes une complexité constante :

- `[]` crée une liste vide (c'est-à-dire ne contenant aucun élément).
- `len(liste)` renvoie la longueur de la liste `liste`.
- `liste.append(x)` ajoute l'élément `x` à la fin de la liste `liste`.
- `liste[i]` renvoie le `(i + 1)`-ième élément de la liste `liste` s'il existe ou produit une erreur sinon (noter que le premier élément de la liste est `liste[0]`).



L'expression `[k for i in range(n)]` construit une liste de longueur `n` contenant `n` occurrences de `k`.

**Important** : L'utilisation de toute autre fonction sur les listes telle que `liste.insert(i,x)`, `liste.remove(x)`, `liste.index(x)`, ou encore `liste.sort(x)` est rigoureusement interdite. Ces fonctions devront être réécrites explicitement si nécessaire.

On rappelle que l'on peut récupérer directement les valeurs contenues dans un tuple de la façon suivante : après l'instruction `a,b,c = (1,2,4)`, la variable `a` contient la valeur 1, `b` contient la valeur 2 et `c` contient la valeur 4. Cette instruction engendre une erreur si le nombre de variables à gauche est différent de la taille du tuple à droite.

Dans la suite, nous distinguerons *fonctions* et *procédures* : les fonctions renvoient une valeur (un entier, une liste, un couple, etc.) tandis que les procédures ne renvoient aucune valeur.

La partie I contient les principales définitions, les parties II, III, IV et V sont indépendantes.

*Nous attacherons la plus grande importance à la lisibilité du code produit par les candidats ; aussi, nous encourageons les candidats à utiliser des commentaires et à introduire des procédures ou des fonctions intermédiaires pour faciliter la compréhension du code.*

## Partie I. Initialisation et affichage de l'aire de jeu

L'aire de jeu est représentée par une grille de dimensions `largeur`×`hauteur`. Dans l'exemple de la Figure 1(a), la grille est de dimensions 6×12.

Chaque case de la grille contient une valeur qui représente soit une case vide, soit une couleur. On utilisera les constantes entières suivantes pour représenter l'état des cases de la grille (une constante est une variable globale qui n'est jamais modifiée après création) :

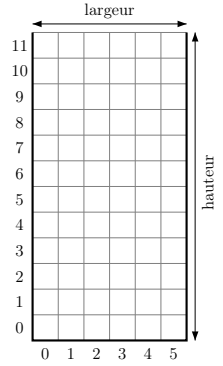
- `VIDE` pour une case vide ;
- `R`, `V`, `B`, `N`, `J` pour les couleurs.

On supposera bien sûr que ces constantes sont deux à deux distinctes et on n'utilisera aucune autre variable globale. La Figure 1(b) montre un exemple d'aire de jeu en cours de partie. Ci-dessous, sa représentation en Python sous la forme d'un tableau de tableaux, ou plus précisément d'un tableau de taille `largeur` (ici 6) contenant dans chaque case une colonne, qui est elle-même un tableau de taille `hauteur` (ici 12) :

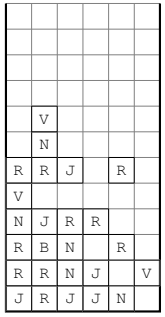
```
grille = [[J,    R,    R,    N,    V,    R,    VIDE, VIDE, VIDE, VIDE, VIDE, VIDE],
          [R,    R,    B,    J,    VIDE, R,    N,    V,    VIDE, VIDE, VIDE, VIDE],
          [J,    N,    N,    R,    VIDE, J,    VIDE, VIDE, VIDE, VIDE, VIDE, VIDE],
          [J,    J,    VIDE, R,    VIDE, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE],
          [N,    VIDE, R,    VIDE, VIDE, R,    VIDE, VIDE, VIDE, VIDE, VIDE, VIDE],
          [VIDE, V,    VIDE, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE]]
```

À noter que la valeur de la case supérieure droite de la grille est `grille[5][11]`. On veillera à respecter l'ordre des dimensions afin que la case de coordonnées $(i, j)$ avec $0 \leq i <$ **`largeur`** et $0 \leq j <$ **`hauteur`** (voir Figure 1(a)) corresponde bien à la valeur `grille[i][j]` dans sa représentation en Python.

**Question 1.** Écrire une fonction `creerGrille(largeur,hauteur)` qui renvoie une grille de dimensions `largeur`×`hauteur` dont toutes les cases sont vides.




(a) Aire de jeu représentée par une grille.




(b) Grille en cours de partie.

FIGURE 1 – L'aire de jeu.



On souhaite pouvoir afficher à l'écran le contenu de l'aire de jeu. Pour cela, on suppose l'existence des procédures suivantes :

- `afficheCouleur(c)` qui prend en argument une constante de couleur `c` $\in \{$`R`, `V`, `B`, `N`, `J`$\}$ et affiche le caractère correspondant ;
- `afficheBlanc()` qui affiche un espace vide ;
- `nouvelleLigne()` qui déplace le curseur au début de la ligne suivante.

**Important** : Il est *rigoureusement interdit* d'utiliser toute autre fonction d'affichage, notamment la commande `print`.

**Question 2.** Écrire une procédure `afficheGrille(grille)` qui affiche à l'écran le contenu de l'aire de jeu, encodé dans le tableau `grille`. On veillera à bien respecter *l'orientation verticale* de la grille : la ligne apparaissant en bas de l'écran doit correspondre aux éléments `grille[i][0]` pour $0 \leq i <$ **`largeur`**. Par exemple, la grille de la Figure 1(b) sera affichée comme le montre la capture d'écran ci-contre.

```
 V
 N
RRJ R
V
NJRR
RBN R
RRNJ V
JRJJN
```

## Partie II. Création et mouvement du barreau

Au cours d'une partie, le joueur voit apparaître à l'écran un *barreau* consistant en une tour de $k$ blocs colorés, où $k \geq 3$. Le barreau apparaît en pointillé sur les figures. La valeur de $k$ et la couleur des blocs du barreau sont choisies aléatoirement. La position d'un barreau est donnée par les coordonnées $(x, y)$ de son bloc inférieur. Dans la suite, on supposera toujours que les coordonnées $(x, y)$ définissent une case dans la grille (c'est-à-dire $0 \leq x <$ `largeur` et $0 \leq y <$ `hauteur`) et que $y + k \leq$ `hauteur` (et donc $k \leq$ `hauteur`).

Un barreau de taille $k$ peut naître au sommet d'une colonne ayant ses $k$ cases supérieures vides (voir Figure 2). Si aucune colonne n'a ses $k$ cases supérieures vides, la partie s'arrête.

**Question 3.** Écrire une fonction `grilleLibre(grille,k)` qui renvoie `True` si dans au moins une colonne de la grille, les `k` premières cases (en partant du haut de la grille) sont vides, et renvoie `False` sinon. On ne fait pas d'hypothèse sur le contenu de la grille (en particulier, la grille n'est pas nécessairement tassée). Quelle est la complexité de votre fonction ?

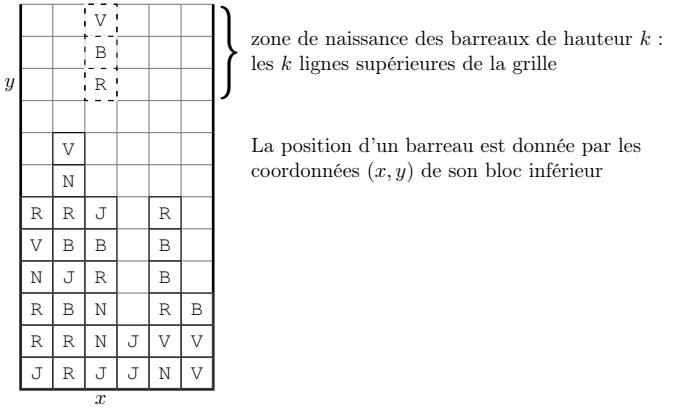



FIGURE 2 – Zone de naissance des barreaux.



En l'absence d'intervention du joueur, le barreau descend d'une case à chaque unité de temps. La descente s'arrête dès que le barreau atteint le bas de la grille, ou rencontre un bloc déjà présent (le passage du temps est illustré à la Figure 3).

**Question 4.** Écrire une procédure `descente(grille,x,y,k)` qui prend en arguments une grille et les coordonnées `(x,y)` du bloc inférieur d'un barreau de hauteur `k`, et modifie la grille en faisant descendre le barreau d'une case. Si le barreau ne peut pas descendre, la grille n'est pas modifiée.

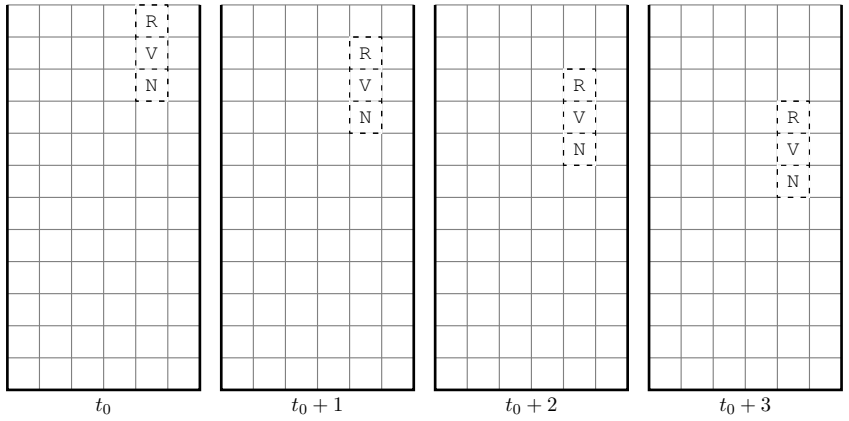



FIGURE 3 – Descente du barreau.

Le joueur peut déplacer le barreau d'une colonne vers la gauche (ou vers la droite) en utilisant les flèches du clavier. Le déplacement du barreau vers la droite n'est possible que si le barreau n'est pas contre le bord droit de la grille et que les $k$ cases se trouvant à sa droite sont toutes vides. Symétriquement, le déplacement du barreau vers la gauche n'est possible que si le barreau n'est pas contre le bord gauche de la grille et que les $k$ cases se trouvant à sa gauche sont toutes vides (Figure 4).

**Question 5.** Écrire une procédure `deplacerBarreau(grille,x,y,k,direction)` qui prend en argument une grille et les coordonnées (`x`,`y`) du bloc inférieur d'un barreau de hauteur `k`, et un entier `direction` $\in \{-1, 1\}$, et qui modifie la grille en déplaçant le barreau d'une case vers la gauche (si `direction` $= -1$) ou vers la droite (si `direction` $= 1$). Si le déplacement du barreau n'est pas possible, la grille reste inchangée.

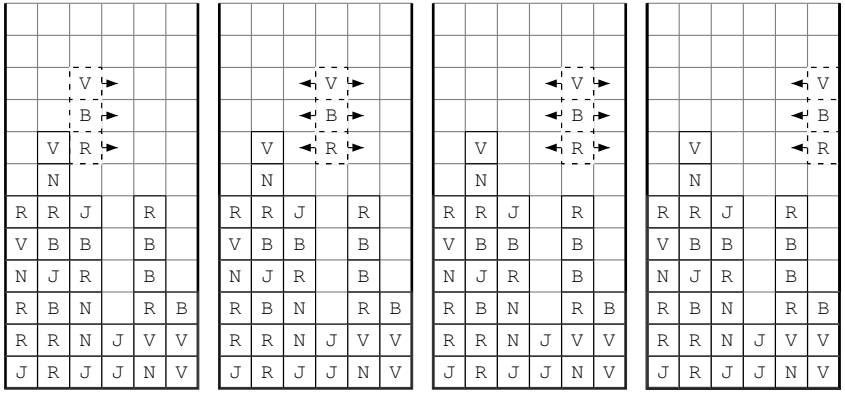


FIGURE 4 – Déplacements possibles du barreau.



Le joueur peut aussi modifier l'ordre des blocs dans le barreau par une permutation circulaire qui fait remonter chaque bloc d'un case, sauf le bloc le plus haut qui redescend à la place du bloc le plus bas (Figure 5).

**Question 6.** Écrire une procédure `permuterBarreau(grille,x,y,k)` qui modifie la grille en effectuant, comme décrit ci-dessus, une permutation circulaire des couleurs du barreau de hauteur `k` dont le bloc inférieur est en position (`x`,`y`) dans la grille donnée en argument.

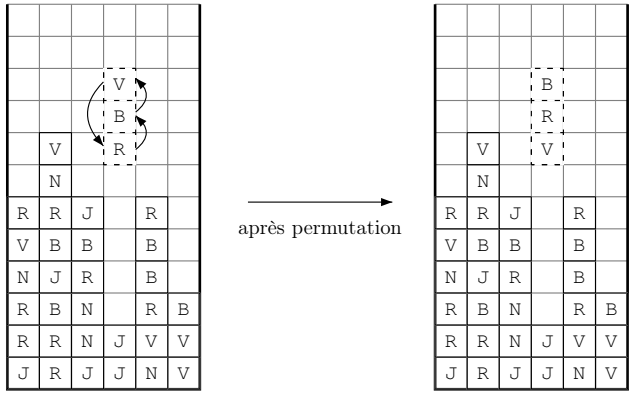



FIGURE 5 – Permutation circulaire du barreau.

Enfin, le joueur peut faire descendre le barreau « rapidement », ce qui signifie que le barreau descend du nombre maximum possible de cases (Figure 6), pour venir reposer au-dessus de la première case non vide de la grille située sous le barreau, ou sur le fond de la grille (si toutes les cases sous le barreau sont vides). Dans l'exemple de la Figure 6, la première case non vide sous le barreau a pour coordonnées $(3, 1)$.

**Question 7.** Écrire une procédure `descenteRapide(grille,x,y,k)` qui prend en argument une grille et les coordonnées (`x`,`y`) du bloc inférieur d'un barreau de hauteur `k`, et modifie la grille en faisant descendre le barreau « rapidement ». On demande que la fonction ait une complexité en $\mathcal{O}(\texttt{k} + \texttt{hauteur})$ (et non $\mathcal{O}(\texttt{k} \times \texttt{hauteur})$).

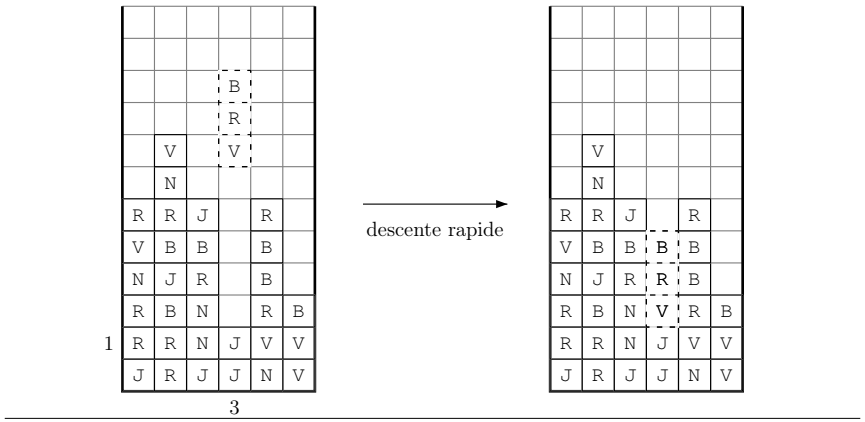



FIGURE 6 – Descente rapide du barreau.



## Partie III. Détection des alignements et calcul du score

Lorsqu'un barreau ne peut plus descendre, le joueur gagne des points si des alignements d'au moins trois blocs de la même couleur sont réalisés dans la grille. Les alignements peuvent être réalisés sur une ligne, sur une colonne, ou en diagonale. Notre but est maintenant de détecter les alignements unicolores et de calculer le score du joueur.

Chaque alignement unicolore de longueur $m \geq 3$ donne $m - 2$ points au joueur. Cette règle ne s'applique que si l'alignement de longueur $m$ n'est pas lui-même inclus dans un alignement de longueur plus grande que $m$, donc on ne prend en considération que les alignements de longueur maximale pour calculer le score. Par exemple l'alignement horizontal de quatre blocs de couleur `B` dans la Figure 7 donne 2 points, ceux en diagonale de trois blocs de couleur `B` et `V` donnent chacun 1 point. Le bloc `B` de coordonnées $(3, 4)$ compte à la fois pour l'alignement horizontal et l'alignement en diagonale. Le joueur marque donc 4 points dans cette configuration.

Tous les blocs appartenant à de tels alignements sont ensuite *simultanément* retirés de la grille et remplacés par des cases vides (deuxième grille de la Figure 7). Les blocs restants sont ensuite tassés, c'est-à-dire qu'ils descendent du maximum possible de cases. Il se peut que de nouveaux alignements se forment après le tassement de la grille (comme les cinq blocs de couleur `R` dans la troisième grille de la Figure 7). Ces nouveaux alignements donnent à nouveau des points au joueur (selon le même barème que précédemment) et le même processus d'élimination des alignements et de tassement de la grille est réalisé. Ce processus se poursuit jusqu'à ce qu'il n'y ait plus d'alignement unicolore de longueur $m \geq 3$ dans la grille. Dans l'exemple, le joueur marque au total 7 points.

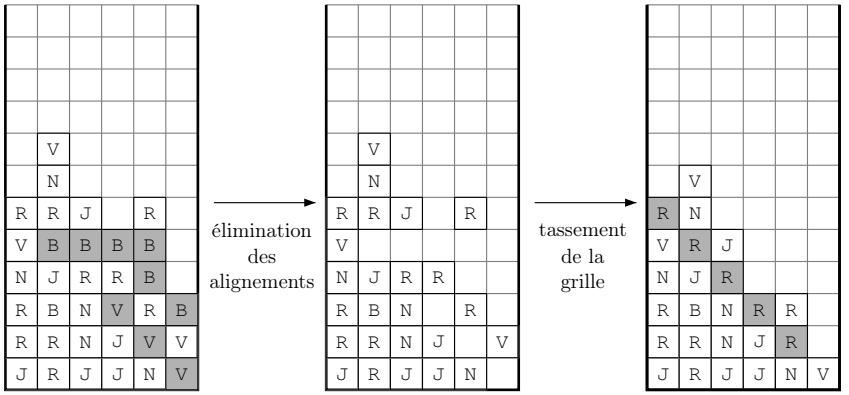
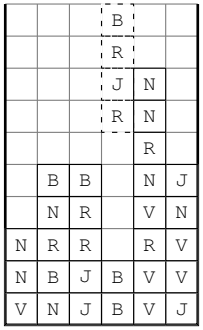



FIGURE 7 – Élimination des alignements unicolores et tassement de la grille.

**Question 8.** Calculer le nombre de points marqués par le joueur s'il laisse descendre le barreau dans l'aire de jeu ci-dessous. Justifier votre réponse.




Pour réaliser la fonction qui va détecter et comptabiliser les alignements unicolores de la grille, on va d'abord construire une fonction qui réalise cela sur un tableau simple (à une dimension).

**Question 9.** Écrire une fonction `detecteAlignement(rangee)` qui prend en argument un tableau `rangee` non vide à une dimension contenant des valeurs dans l'ensemble {`VIDE`, `R`, `V`, `B`, `N`, `J`}, et qui renvoie un tuple `(marking, score)` de deux éléments :

- `marking` est un tableau de la même taille que `rangee` contenant des Booléens, tel que `marking[i] = True` si et seulement si `rangee[i]` appartient à un alignement unicolore de longueur au moins 3.
- `score` est le nombre de points obtenus par le joueur pour les alignements présents dans `rangee` (selon le barème donné plus haut).

Par exemple, pour le tableau `rangee = [B, R, R, R, R, J, J, J, VIDE, VIDE, VIDE]`, la fonction `detecteAlignement(rangee)` renvoie :

- `marking = [False,True,True,True,True,True,True,True,False,False,False]`
- `score = 3`.

On demande que lors du traitement, la fonction `detecteAlignement(rangee)` n'accède qu'une seule fois à chaque élément du tableau `rangee`.

**Indice :** Parcourir le tableau et détecter les changements de couleur.



Pour détecter les alignements unicolores de la grille et comptabiliser les points marqués, nous allons explorer toutes les lignes, colonnes et diagonales de la grille. Pendant le traitement, on va conserver une copie de travail de la grille dans laquelle on remplacera les cases appartenant à un alignement unicolore par des cases vides. Dans la question suivante, on suppose que `g` est cette copie de travail.

**Question 10.** Écrire une fonction `scoreRangee(grille, g, i, j, dx, dy)` qui prend en argument une grille `grille` et une grille `g` de même dimensions que `grille`, les coordonnées $(i,j)$ d'une case dans la grille et une direction $(dx,dy)$ donnée par $dx,dy \in \{-1,0,1\}$. La fonction construit un tableau `rangee` à une dimension contenant les valeur des cases de `grille` dont les coordonnées sont $(i,j)$, $(i+dx,j+dy)$, $(i+2dx,j+2dy)$, etc. puis utilise la fonction `detecteAlignement(rangee)` de la Question 9 pour détecter les alignements unicolores dans le tableau `rangee`, déterminer le nombre de points marqués et mettre à jour la grille $g$ pour que les cases appartenant à un alignement unicolore dans `grille` soient vides dans $g$. La grille `grille` ne doit pas être modifiée. La fonction renvoie le nombre de points marqués.

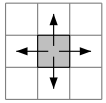
**Question 11.** Écrire une fonction `effaceAlignement(grille)` qui prend en argument une grille et renvoie un tuple `(g, score)` de deux éléments :

- `g` est la grille mise à jour où tous les blocs appartenant à un alignement unicolore sont remplacés par des cases vides.
- `score` est le nombre total de points obtenus par le joueur pour les alignements présents dans la grille.

La grille `grille` ne doit pas être modifiée. Donner la complexité de votre fonction.

**Question 12.** Écrire une procédure `tassementGrille(grille)` qui modifie la grille donnée en argument en effectuant le tassement de ses cases non vides.

**Question 13.** Écrire une fonction `calculScore(grille)` qui met à jour la grille `grille` après élimination des alignements et tassement, répétés jusqu'à ce que la grille ne contienne plus aucun alignement unicolore de longueur $m \geq 3$, et qui renvoie le nombre total de points marqués par le joueur pour les alignements éliminés de la grille.



(a) Voisins d'une case.

| | | | | | |
|---|---|---|---|---|---|
| | | | | | R |
| R | R | | | R | N |
| V | R | R | R | R | J |
| J | V | R | J | R | J |
| J | R | R | J | R | V |
| N | J | R | R | R | J |

(b) Une région unicolore maximale.

FIGURE 8 – Régions unicolores (Partie IV).

## Partie IV. Variante du jeu : régions unicolores

Dans cette partie, on considère une variante du jeu où le but du joueur est de former des régions unicolore au lieu d'alignements (par exemple un carré de $2 \times 2$ cases de la même couleur).

Une région unicolore est un ensemble de cases, toutes de la même couleur, qui est connexe pour la relation d'adjacence suivante : deux cases sont adjacentes si elles partagent un côté (Figure 8(a)).

La région grisée de la Figure 8(b) est une région unicolore maximale : si on y ajoute une case quelconque, elle ne reste plus connexe et unicolore. En particulier, la case supérieure droite ne peut être ajoutée car elle n'est adjacente à aucune case de la région grisée.

**Question 14.** Écrire une fonction *récursive* `tailleRegionUnicolore(grille,x,y)` qui renvoie le nombre de cases appartenant à la plus grande région unicolore de la grille contenant la case (x,y). Justifier la terminaison de votre fonction.

Considérons le code Python aux Figure 9 et Figure 10, dont le but est de réaliser le même travail que la fonction `tailleRegionUnicolore` de la Question 14 sans utiliser la récursivité.

Intuitivement, la fonction `exploreRegion(grille,x,y)` effectue un balayage de la grille, d'abord verticalement à partir de la case (x,y), puis verticalement à partir de chaque voisine horizontale des cases déjà explorées.

**Question 15.** Déterminer si la fonction `exploreRegion(grille,x,y)` renvoie le nombre de cases appartenant à la plus grande région unicolore de la grille contenant la case (x,y).

Si oui, justifier soigneusement que la fonction renvoie toujours une valeur correcte.

Si non, donner un exemple de paramètres `grille`, `x`, `y` pour lesquels la valeur renvoyée par la fonction est incorrecte (on pourra dessiner la grille).



```python
def xDansGrille(grille, x):
  largeur = len(grille)
  return (x >= 0) and (x < largeur)
```

```python
def yDansGrille(grille, y):
  hauteur = len(grille[0])
  return (y >= 0) and (y < hauteur)
```

```python
def exploreVertical(grille, x, y, dir):
  hauteur = len(grille[0])
  couleur = grille[x][y]
  v = y + dir

  while yDansGrille(grille, v):
    if grille[x][v] != couleur:
       return v - dir
    v = v + dir

  if dir == 1:
    return hauteur - 1
  else:
    return 0
```

```python
def exploreRegion(grille, x, y):

  inf = exploreVertical(grille, x, y, -1)   # explore vers le bas
  sup = exploreVertical(grille, x, y, 1)    # explore vers le haut

  d = exploreHorizontal(grille, x, y, 1)    # explore vers la droite
  g = exploreHorizontal(grille, x, y, -1)   # explore vers la gauche

  score = sup - inf + 1 + d + g
  return score
```

FIGURE 9 – Code Python pour la Question 15.

```python
def exploreHorizontal(grille, x, y, dir):
  largeur = len(grille)
  couleur = grille[x][y]

  inf = exploreVertical(grille, x, y, -1)   # explore vers le bas
  sup = exploreVertical(grille, x, y, 1)    # explore vers le haut

  score = 0
  u = x + dir

  while xDansGrille(grille, u) and (inf <= sup):
    v = inf
    infNew = 1        # initialement, infNew > supNew
    supNew = 0

    while (v <= sup):
      if grille[u-dir][v] == couleur and grille[u][v] == couleur:
         infNew = exploreVertical(grille, u, v, -1)   # explore vers le bas
         supNew = exploreVertical(grille, u, v, 1)    # explore vers le haut
         score = score + supNew - infNew + 1
         v = supNew + 2

         while (v <= sup):
            if grille[u-dir][v] == couleur and grille[u][v] == couleur:
               supNew = exploreVertical(grille, u, v, 1)  # explore vers le haut
               score = score + supNew - v + 1
               v = supNew + 1
            v = v + 1
      v = v + 1

    inf = infNew
    sup = supNew
    u = u + dir

  return score
```

FIGURE 10 – Code Python pour la Question 15.



## Partie V. Gestion des scores en SQL

On souhaite utiliser une base de données pour stocker les résultats obtenus par une communauté de joueurs. On suppose que l'on dispose d'une base de données comportant les tables `JOUEURS(id_j, nom, pays)` et `PARTIES(id_p, date, duree, score, id_joueur)` où :

- `id_j`, de type entier, est la clé primaire de la table `JOUEURS`,
- `nom` est une chaîne de caractères donnant le nom du joueur,
- `pays` est une chaîne de caractères donnant le pays du joueur,
- `id_p`, de type entier, est la clé primaire de la table `PARTIES`,
- `date` est la date (`AAAAMMJJ`) de la partie,
- `duree`, de type entier, est la durée en secondes de la partie,
- `score`, de type entier, est le nombre de points marqués au cours de la partie,
- `id_joueur` est un entier qui identifie le joueur de la partie.

**Question 16.** Étant donné une chaîne de caractères `cc` contenant le nom d'un joueur, écrire une requête SQL qui renvoie la date, la durée et le score de toutes les parties jouées par le joueur `cc`, listées par ordre chronologique (au choix, croissant ou décroissant).

**Question 17.** Étant donné un entier $s$ (le score que vient de réaliser une joueuse nommée Alice), écrire une requête SQL qui renvoie la position qu'aura le score $s$ dans le classement des parties par ordre de score (on suppose que la dernière partie d'Alice n'a pas encore été insérée dans la table des parties). En cas d'ex aequo pour le score $s$ (une ou plusieurs parties déjà présentes ayant le score $s$), le rang sera le même que s'il n'y avait qu'une seule partie avec le score $s$.

Par exemple, la requête renverra 1 (le score d'Alice est "1$^{\text{er}}$") si aucun score n'est meilleur que $s$. Autre exemple, si la base de données contient 6 parties dont les scores sont $87, 75, 75, 63, 60, 60$, alors le rang de $s = 75$ sera 2, le rang de $s = 70$ sera 4 et le rang de $s = 60$ sera 5.

**Question 18.** Écrire une requête SQL qui renvoie le record de France de Tetris couleur, c'est-à-dire le meilleur score réalisé par un joueur dont le pays est la France.

**Question 19.** Étant donné une chaîne de caractères `cc` contenant le nom d'un joueur (ayant déjà joué au moins une partie de Tetris couleur), écrire une requête SQL qui renvoie le rang du joueur `cc`, c'est-à-dire sa position dans le classement des joueurs par ordre de leur meilleur score dans une partie de Tetris couleur (on traitera les ex aequo de la même manière qu'à la question 17).

F I N

* *
*

## Partie I.

**Question 1.** On fait attention à bien créer une nouvelle liste à chaque étape.

```
def creerGrille(largeur, hauteur):
    Grille=[]
    for i in range(largeur):
        Grille.append([VIDE for j in range(hauteur)])
    return Grille
# Version alternative ; là encore on crée une nouvelle liste "colonne"
def creerGrille_alternative(largeur, hauteur):
    return  [[VIDE for j in range(hauteur)] for i in range(largeur)]
```

**Question 2.** *Je n'ose pas utiliser les indexations négatives comme* `Grille[-1][-1]`

```
def afficheGrille(grille):
    largeur=len(grille)
    hauteur=len(grille[0])
    for j in range(hauteur):
        if j!=0: nouvelleLigne()
        for i in range(largeur):
            c=grille[i][hauteur-j-1]
            if c== VIDE:
                afficherblanc()
            else:
                afficherCouleur(c)
```

## Partie II.

**Question 3.** On teste les $k$ dernière cases pour chaque colonne. On renvoie `True` dès qu'une colonne convient. Si aucune ne convient, on renvoie `False`

```
def grilleLibre(grille,k):
    largeur=len(grille)
    hauteur=len(grille[0])
    x=0
    while x < largeur:
        y=hauteur-k
        while y<hauteur and grille[x][y]==VIDE:
            y+=1
        if y==hauteur:
            return True
        x+=1
    return False
```

La boucle interne comporte au maximum $k$ itérations. La boucle externe comporte au maximum `largeur` itérations.

Chaque boucle s'effectue en temps constant. Ainsi la complexité est en $\mathrm{O}(k \times \texttt{largeur})$.

**Question 4.** Si la case sous la tour existe et est vide, on fait descendre la tour case par case en commençant par le bas.

```
def descente(grille,x,y,k):
    if y!=0 and grille[x][y-1]==VIDE:
        for j in range(y,y+k):
            grille[x][j-1]=grille[x][j]
        grille[x][y+k-1]=VIDE
```





**Question 5.** On commence par tester la possibilité de déplacement par rapport aux bords de la grille puis on vérifie si les cases sont vides. Une fois ces vérifications effectuées on effectue le déplacement

```
def deplacerBarreau(grille,x,y,k,direction):
    largeur=len(grille)
    if 0<=x+direction and x+direction < largeur: #test bords
        j=y
        while j<y+k and grille[x+direction][j]==VIDE:
            j+=1
        if j == y+k: # la destination est complètement vide
            for j in range(y,y+k):
                grille[x+direction][j]=grille[x][j]
                grille[x][j]=VIDE
```

**Question 6.** On stocke la case du haut dans une variable tampon et on déplace de haut en bas.

```
def permuterBarreau(grille,x,y,k):
    tampon=grille[x][y+k-1]
    for j in range(y+k-1,y,-1):
        grille[x][j]=grille[x][j-1]
    grille[x][y]=tampon
```

**Question 7.** On peut être tenté (épreuve en temps limité) de réitérer `descente` tant que c'est possible.

C'est pour cela que le sujet donne la contrainte sur la complexité.

La boucle conditionnelle (`while`), permet de trouver le pas de la descente et a une complexité en $\mathrm{O}(\texttt{hauteur})$ puis la boucle inconditionnelle (`for`) qui suit a une complexité en $\mathrm{O}(k)$, les cases y sont descendues une à une sans risque de "chevauchement".

```
def descenteRapide(grille,x,y,k):
    pas=0
    while y>=pas+1 and grille[x][y-pas-1]==VIDE:
        pas+=1
    if pas !=0:
        for j in range(k):
            grille[x][y+j-pas]=grille[x][y+j]
            grille[x][y+j]=VIDE
```

## Partie III.

**Question 8.** On obtient les trois configurations suivantes

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | N | |
| | | | | N | |
| | | | B | **R** | |
| | B | B | **R** | N | J |
| | N | **R** | J | V | N |
| N | **R** | **R** | **R** | **R** | V |
| N | B | J | B | V | V |
| V | N | J | B | V | J |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | **N** | |
| | | | | **N** | J |
| | **B** | | **B** | **N** | N |
| N | N | **B** | J | **V** | V |
| N | **B** | J | **B** | **V** | V |
| V | N | J | B | **V** | J |

et enfin

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | J |
| | | | | | N |
| N | | | | | V |
| N | N | J | J | | V |
| V | N | J | B | | J |

Ce qui donne $(2+2)+(1+1+1+1)=8$ points.

**Question 9.** On parcourt `rangee` en une fois. On note la couleur précédente (ou éventuellement `VIDE`) dans la variable `couleur`. Si la couleur est identique à la précédente, on incrémente la variable `lbloc`.
Si la couleur est distincte mais que `lbloc>2`, on met à jour la variable entière `score` et le tableau `marking`.
En fin de boucle, on traite l'éventuel dernière répétition de couleurs.

```
def detecteAlignement(rangee):
    n=len(rangee)
    marking=[False for j in range(n)]
    score=0
    couleur=rangee[0]
    lbloc=1
    for i in range(1,n):
        if rangee[i]==VIDE or rangee[i]!=couleur:
            if lbloc>2:
                score+=lbloc-2
                for j in range(lbloc):
                    marking[i-1-j]=True
            lbloc=1
            couleur=rangee[i]
        else:
            lbloc+=1
    if lbloc>2:
        score+=lbloc-2
        for j in range(lbloc):
            marking[n-1-j]=True
    return (marking, score)
```

**Question 10.** Il suffit d'appliquer les consignes.

```
def scoreRangee(grille, g, i, j, dx, dy):
    largeur=len(grille)
    hauteur=len(grille[0])
    x,y=i,j
    rangee=[]
    while 0 <= x and x<largeur and 0<=y and y < hauteur:
        rangee.append(grille[x][y])
        x+=dx
        y+=dy
    score,marking=detecteAlignement(rangee)
    n=len(rangee)
    for p in range(n):
        if marking[p]:
            g[i+p*dx][j+p*dy]=VIDE
    return score
```





**Question 11.** On commence par copier la grille. Pour le parcours des rangées, on remarque que le "vecteur" `(dx,dy)` dirige les mêmes rangées que le "vecteur" `(-dx,-dy)`.

```
def copie(grille):
    """fonction qui réalise une VRAIE copie de grille"""
    g=[]
    for x in range(len(grille)):
        colonne=[]
        for y  in range(len(grille[0])):
            colonne.append(grille[x][y])
        g.append(colonne)
    return g
# copie alternative
def copie_alternative(grille):
    return [[grille[i][j] for j in range(hauteur)] for i in range(largeur)]

def effaceAlignement(grille):
    g=copie(grille)
    largeur=len(grille)
    hauteur=len(grille[0])
    score=0
    dx,dy=1,1
    for i in range(largeur):
        score+= scoreRangee(grille, g, i, 0, dx, dy)
    for j in range(1,hauteur):
        score+= scoreRangee(grille, g, 0, j, dx, dy)
    dx,dy=1,0
    for j in range(hauteur):
        score+= scoreRangee(grille, g, 0, j, dx, dy)
    dx,dy=1,-1
    for i in range(largeur):
        score+= scoreRangee(grille, g, i, hauteur-1, dx, dy)
    for j in range(hauteur-1):
        score+= scoreRangee(grille, g, 0, j, dx, dy)
    dx,dy=0,1
    for i in range(largeur):
        score+= scoreRangee(grille, g, i, 0, dx, dy)
    return (g,score)
```

La fonction `copie` a une complexité en O(`hauteur*largeur`)
La fonction `scoreRangee` si on itère en `largeur` (respectivement en `hauteur`) a une complexité en O(`hauteur+largeur`) (respectivement en O(`largeur`))
donc chacune de ces boucles a une complexité en O(`hauteur*largeur`)
En conclusion la complexité de la fonction `effaceAlignement` est en O(`hauteur*largeur`).

**Question 12.** Sur chaque colonne, on pratique la "descente rapide" en partant du bas par blocs constitués d'une seule case. La valeur de la variable `pas` est un entier donnant le nombre de cases vides rencontrées dans la colonne.

```
def tassementGrille2(grille):
    largeur=len(grille)
    hauteur=len(grille[0])
    for x in range(largeur):
        pas=0
        for y in range(hauteur):
            if grille[x][y]==VIDE:
                pas+=1
            elif pas !=0:
                grille[x][y-pas]=grille[x][y]
                grille[x][y]=VIDE
```

**Question 13.** On utilise deux variables pour la grille `g1` avant traitement et `g2` après. On s'arrête lorsqu'il n'y a plus d'évolutions .

```
def calculScore(grille):
    largeur=len(grille)
    hauteur=len(grille[0])
    g1=grille
    g2,score= effaceAlignement(g1)
    tassementGrille(g2)
    test=score
    while test !=0: #la grille a changé
        g1=g2
        g2,test= effaceAlignement(g1)
        tassementGrille(g2)
        score+= test
    for x in range(largeur):
        for y in range(hauteur):
            grille[x][y]=g2[x][y]
    return score
```

## Partie IV.

**Question 14.** J'utilise une fonction auxiliaire qui travaille sur une copie de la grille. On ne parcourt que les cases de la composante connexe et à chaque appel récursif on met la case à vide.

Ainsi le nombre d'appel de la fonction récursive auxiliaire est égal au cardinal de la composante connexe parcourue.

Ainsi à chaque appel récursif, on a une suite d'entiers strictement décroissante ce qui justifie la terminaison.

Dans la fonction auxiliaire, les variables `g` (copie du graphe) et `c` (couleur de la composante connexe) sont considérées comme globales voire non locales.

En ce qui concerne la correction, on peut le faire par récurrence forte (ou totale ou avec prédécesseurs) sur le cardinal de la composante connexe. Il suffit de remarquer que la composante chromatique connexe d'un point est la réunion disjointe du singleton formé par ce point ainsi que des composantes connexes de ses voisins dans le graphe auquel le point a été décoléré.





```
def tailleRegionUnicolore(grille,x,y):
    g=copie(grille)
    largeur=len(g)
    hauteur=len(g[0])
    c=g[x][y]
#on ne vérifie pas s'il s'agit d'une couleur
#confiance totale en l'énoncé
    def auxrec(x,y):
        g[x][y]= VIDE
        res=1
        if x>0 and g[x-1][y]==c:
            res+= auxrec(x-1,y)
        if x+1<largeur and g[x+1][y]==c:
            res+= auxrec(x+1,y)
        if y>0 and g[x][y-1]==c:
            res+= auxrec(x,y-1)
        if y+1<hauteur and g[x][y+1]==c:
            res+= auxrec(x,y+1)
        return res
    return auxrec(x,y)
```

**Question 15.** La stratégie proposée ne fonctionne pas.

Par exemple avec `grille=[[J,V,J],[J,J,J]]`

On n'atteint pas le `J` de `grille[0][2]` en partant de la case `grille[0][0]`.

D'ailleurs en faisant l'essai `exploreRegion(grille, 0, 0)` le programme renvoie `4` alors que `tailleRegionUnicolore(grille,0,0)` renvoie `5` comme prévu.

## Partie V.

**Question 16.**

```
SELECT date,duree,score FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
WHERE nom=cc ORDER BY date;
```

**Question 17.**

```
SELECT 1+COUNT(*) FROM PARTIES WHERE score>s;
```

**Question 18.**

```
SELECT MAX(score) FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
WHERE pays='France';
```

**Question 19.**

```
SELECT   COUNT(*) FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
GROUP BY id_j HAVING MAX(score) >
(SELECT MAX(score) FROM JOUEURS JOIN PARTIES ON id_j=id_joueur WHERE nom=cc);
```

* *
*

ÉCOLE POLYTECHNIQUE CONCOURS D'ADMISSION 2019
ENS : LYON, PARIS-SACLAY, RENNES

# Composition d'informatique 2h, filière MP (XLCR)

## Bilan général *(Candidats français)*

| | | |
|---|---|---|
| 0<=N<4 | 22 | 11,7% |
| 4<=N<8 | 94 | 50% |
| 8<=N<12 | 64 | 34,04% |
| 12<=N<16 | 8 | 4,26% |
| 16<=N<=20 | 0 | 0 |
| Total : | 188 | 100% |
| Nombre de candidats : | 188 | |
| Note moyenne : | 7,15 | |
| Ecart-type : | 2,67 | |

## Commentaires

Cette année, le sujet portait sur la réalisation d'un jeu de type Tetris, où un joueur fait tomber un barreau coloré dans une grille, dans le but de réaliser des alignements monochromes horizontaux les plus grandes possibles. La première partie portait sur l'initialisation et l'affichage d'une grille. La seconde partie se concentrait sur les déplacements possibles du barreau. Dans la troisième partie il était demandé d'implémenter la détection d'alignements de couleurs et le calcul du score. La quatrième partie considérait une variante plus technique, où les régions monochromes étaient décomptées. Enfin, la cinquième partie étudiait la gestion d'une base de données regroupant les scores des parties effectuées.

Le langage Python utilisant l'indentation comme marqueur de blocs, il est rappelé -- comme chaque année -- de bien veiller à ce que cette indentation soit la plus claire possible: soit en suivant scrupuleusement le quadrillage de la copie (compter au moins 1cm par indentation), soit en traçant des repères verticaux au niveau de chaque tabulation pour lever tout doute. Attention dans ce cas : bien veiller à ne pas omettre de bloc, et à ne pas inclure la ligne "d'en-tête" du bloc, qui ne doit pas être indentée.

L'énoncé décrivait précisément les opérations autorisées sur les listes : création de liste vide, `len`, `append`, accès via `L[i]` pour $0 \leq i <$ len(L), et compréhension de liste `[ … for x in …. ]`. Les opérations non-autorisées incluaient donc:

- les tranches de listes `L[i:j]`
- la copie de liste via `list.copy` ou `L[:]` (qui par ailleurs étaient souvent mal comprises dans les tableaux à deux dimensions)
- les accès par le dernier élément `L[-1]`
- la multiplication `*` pour recopier une liste
- l'égalité entre les listes



De nombreuses copies ont fait preuve d'inattention avec les éléments basiques du langage: parenthèses aux appels de fonction ; deux-points en fin de `if`, `for`, ou `while` ; oubli de l'opérateur `*` pour la multiplication ; `==` pour la comparaison ; distinction majuscule/minuscule (en particulier pour la constante "`VIDE`"), etc. Ces erreurs traduisent souvent une difficulté à écrire un code correct de soi-même, sans avoir à se reposer sur un éditeur ou la sortie du compilateur qui énumèrent les imprécisions.

Une autre source de confusion portait sur les paramètres et valeurs de retour des fonctions et procédures. La plupart travaillaient sur une grille passée en paramètre, et il était parfois demandé de modifier la grille en place, ou bien de travailler sur une copie et de la renvoyer: une mauvaise compréhension de la consigne entraînait au mieux des `return` superflus, au pire une destruction de la grille originale.

Il était rappelé l'utilité des commentaires, en particulier en introduction de fonctions, pour clarifier le code et l'algorithme employé. Certains sont tombés dans l'excès inverse, en paraphrasant l'intégralité de leur code dans un long commentaire en début de chaque question: il est préférable de focaliser les commentaires sur les points essentiels.

Enfin, on rencontre régulièrement des instruction redondantes, du type "`if a == true:`", ou bien un `else if` qui reprend exactement la négation de la première condition. Cela ne représente pas des erreurs à proprement parler, mais traduisent un manque de recul sur le code que l'on est en train d'écrire.

## Commentaires détaillés

Pour chacune des questions, sont indiquées les statistiques suivantes:

- Traitement: taux de traitement de la question par les candidats (les autres lignes se rapportent uniquement aux copies ayant traité la question)
- Moyenne: note moyenne pour cette question, ramenée sur 1
- 0, ]0 ; 0,5[, [0,5; 1[, 1 : pourcentage de copies dont la note est dans l'intervalle correspondant

### Partie 1

Cette partie avait pour vocation de familiariser le candidat avec les grilles manipulées, via la réalisation de deux fonctions simples (initialisation et affichage de la grille). Pour ces fonctions simples, il était important de bien veiller à respecter les bases du langage.

#### Question 1

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 100 % | 0,65 | 34 % | 0 % | 1 % | 65 % |

L'erreur la plus fréquente consistait à réutiliser une même ligne répétée `hauteur` fois (ces fonctions avaient généralement une complexité en O(hauteur+largeur), alors qu'il y a hauteur*largeur cases à créer, et que les fonctions autorisées ne permettaient de créer les cases qu'une par une. Certains candidats ont également inversé les axes de la grille. La solution la plus efficace (et la plus courte) consistait à utiliser deux compréhensions de listes imbriquées.

### Question 2

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 99 % | 0,54 | 44 % | 0 % | 3 % | 53 % |

Il s'agissait de réaliser deux boucles for imbriquées: une boucle pour les lignes contenant une boucle pour les colonnes, affichant les cases une à une. Il fallait être attentif aux indices, et à l'ordre vertical, l'ordre des lignes de la grille étant inversé par rapport à l'ordre des lignes des fonctions d'impression à l'écran.

## Partie 2

### Question 3

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 100 % | 0,68 | 15 % | 4 % | 31 % | 50 % |

Cette question ne posait pas de piège particulière (en dehors d'un besoin de précision syntaxique). La solution à cette question pouvait être simplifiée par l'utilisation d'une fonction annexe qui teste si les cases sont libres à des coordonnées données (une telle fonction pouvait être réutilisée dans la question 5). Le calcul de la complexité a régulièrement été oublié alors qu'il était correct pour la grande majorité des candidats qui l'ont réalisé.

### Question 4

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 100 % | 0,66 | 16 % | 3 % | 35 % | 46 % |

Deux vérifications étaient attendues avant de réaliser la descente: y>0 d'une part, et que la case en dessous était libre d'autre part (l'ordre des vérifications ayant une importance). Après avoir recopié les éléments un par un, il importait d'effacer l'élément du haut.

### Question 5

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 100 % | 0,74 | 9 % | 3 % | 32 % | 55 % |

Pour cette question il fallait vérifier que la colonne de destination (x+direction) était dans le bon intervalle et contenait bien des cases libres. Beaucoup de candidats ont perdu du temps en traitant les deux directions séparément, au lieu de simplement recopier les cases de la colonne x vers la colonne s+direction.

### Question 6

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 100 % | 0,54 | 44 % | 0 % | 4 % | 52 % |

Deux approches étaient possibles pour cette question. La première consiste à sauvegarder la couleur du barreau supérieur, recopier chaque couleur une case plus haut (de



haut en bas), puis réinsérer la sauvegarde en bas. La seconde consiste à exécuter des inversions deux à deux, soit avec des éléments adjacents (de haut en bas), soit entre l'élément supérieur et chaque autre élément successivement, de bas en haut.

### Question 7

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 99 % | 0,57 | 8 % | 28 % | 40 % | 25 % |

Cette fonction supposait de d'abord rechercher la case de destination, avant de réaliser la descente du barreau sans oublier d'effacer l'original. La recherche de la première case non-vide devait se faire de haut en bas, pour gérer le cas où la grille n'est pas tassée. Deux autres cas spécifiques ont souvent été oubliés: la colonne est entièrement vide (la recherche de la première case non-vide se poursuit alors dans les lignes négatives), ou bien il n'y a pas de case libre, auquel cas la destination est identique au point de départ, et l'effacement de l'original se solde souvent par l'effacement complet du barreau.

## Partie 3

### Question 8

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 99 % | 0,43 | 50 % | 0 % | 13 % | 36 % |

Cette question en apparence simple demandait une attention toute particulière pour ne rater aucun alignement (pour un total de 8 points). La grande majorité des erreurs sont dues à des fautes d'inattention, plutôt qu'à une mauvaise compréhension de l'énoncé. La justification la plus efficace consistait à dessiner les grilles successives après chaque tassement, en identifiant bien les alignements réalisés.

### Question 9

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 96 % | 0,50 | 15 % | 22 % | 49 % | 15 % |

Il était possible de réaliser cette question en une passe (recherche des changements de couleurs et calcul du score à la volée), ou bien deux (enregistrement des différences de couleurs dans un tableau, puis parcours de ce tableau). Les deux principaux écueils étaient la gestion de la dernière séquence (qui devait généralement être traitée à part, à l'extérieur de la boucle), et des séquences vides (qui étaient trop souvent comptées comme des séquences à part entière).

### Question 10

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 89 % | 0,68 | 8 % | 22 % | 26 % | 44 % |

Une erreur presque universelle pour cette question était une mauvaise gestion du cas dx=dy=0: dans la grande majorité des cas ces paramètres menaient à une boucle infinie.

### Question 11

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 81 % | 0,41 | 26 % | 27 % | 36 % | 11 % |

La principale difficulté était de s'assurer de parcourir toutes les lignes, les colonnes et les diagonales sans oubli ni répétition. Par ailleurs, il était demandé de ne pas modifier la grille passée en paramètre ; la première étape consistait donc à en faire une copie profonde, ce qui a posé quelques difficultés aux candidats. Remarque: même si l'opérateur de copie "[:]" avait été autorisé, grille[:][:] ne permet pas de faire une copie complète d'un tableau à deux dimensions. La compréhension de liste est la solution la plus simple pour ce problème.

### Question 12

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 67 % | 0,54 | 15 % | 22 % | 47 % | 16 % |

Le tassement d'une colonne pouvait se faire en la remplissant avec les cases non vides rencontrées lors du parcours effectué en remontant depuis le bas. Les tentatives de réutiliser descente ou descenteRapide menaient au mieux à des solutions de complexité sous-optimale.

### Question 13

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 64 % | 0,53 | 18 % | 10 % | 44 % | 27 % |

Cette question ne posait pas de difficulté particulière ; il fallait veiller à renvoyer la somme des scores incrémentaux renvoyés par effaceAlignement lors des appels alternés à effaceAlignement et tassementGrille.

## Partie 4

### Question 14

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 38 % | 0,47 | 29 % | 17 % | 33 % | 20 % |

Pour proposer une solution récursive, il était nécessaire de modifier la grille en effaçant les cases participant d'une région unicolore lors du parcours. La justification de la terminaison, bien qu'elle ne posait pas de difficulté, n'a pas toujours été fournie. Comme c'est souvent le cas avec les fonctions récursives, certains candidats ont tenté des solutions inutilement compliquées où chaque appel récursif était précédé d'un grand nombre de tests. Ici, faire les vérifications sur les coordonnées et les couleurs en début de fonction augmentait grandement la simplicité de la solution.



### Question 15

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 13 % | 0,39 | 40 % | 21 % | 15 % | 24 % |

La fonction exploreRegion présentait plusieurs écueils, que l'on pouvait mettre en lumière à l'aide d'un contre-exemple. Il fallait veiller à bien spécifier le contre-exemple (grille et case de départ).

## Partie 5

### Question 16

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 68 % | 0,85 | 7 % | 2 % | 13 % | 78 % |

Il fallait utiliser une jointure, et ne pas oublier de trier les résultats par ordre chronologique (ORDER BY).

### Question 17

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 53 % | 0,76 | 5 % | 11 % | 29 % | 55 % |

L'erreur la plus fréquente a été un décalage de 1 dans le rang.

### Question 18

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 55 % | 0,88 | 4 % | 7 % | 12 % | 77 % |

Cette question ne posait pas de difficulté particulière, il fallait faire une jointure et faire appel à la fonction MAX().

### Question 19

| Traitement | Moyenne | 0 | ]0;0.5[ | [0.5;1[ | 1 |
|---|---|---|---|---|---|
| 38 % | 0,59 | 12 % | 23 % | 35 % | 30 % |

On pouvait ici utiliser une requête auxiliaire (qui correspondait en réalité à la requête de la question précédente).