X/ENS Informatique B MP-PC-PSI 2019 — Corrigé

Ce corrigé est proposé par Cyril Ravat (professeur en CPGE) ; il a été relu par William Aufort (professeur en CPGE) et Benjamin Monmege (enseignant-chercheur à l'université).

Ce sujet a pour objectif la réalisation de fonctions qui pourraient se trouver à la base d'un jeu de stratégie, dans lequel un barreau vertical de k cases colorées descend, guidé par le joueur, comme dans le célèbre jeu Tetris. Lorsqu'il touche le sol, les alignements de cases de même couleur sont supprimés et des points sont marqués. Si l'interface graphique et les actions du joueur ne sont pas abordées, le sujet traite néanmoins un grand nombre de problématiques telles que l'initialisation du jeu, les déplacements, la détection d'alignements, les suppressions et le comptage des points. L'étude est divisée en cinq parties de tailles variées.

- La première partie étudie l'initialisation de la grille et son affichage en deux questions plutôt abordables.
- On traite ensuite l'apparition du barreau et ses différents mouvements : descente case par case ou plus rapide, déplacement latéral, permutation des cases. Les questions sont d'un niveau intermédiaire et contiennent peu de pièges. Elles demandent avant tout une manipulation correcte de la liste de listes qui modélise la grille du jeu.
- La troisième partie, plus longue que les autres, constitue le cœur de l'épreuve et nous fait inspecter une grille où un barreau vient de toucher le sol. Il faut détecter les alignements de couleurs, les supprimer, faire descendre les cases qui se trouvent alors suspendues et ainsi de suite. Les fonctions à produire sont plus longues et plus élaborées.
- La quatrième partie n'est constituée que de deux questions, traitant une variante bien plus complexe de détection des alignements. Ces questions sont plus difficiles car moins directives. La deuxième en particulier requiert l'analyse assez fine d'un code de plus de cinquante lignes, peu envisageable en moins de dix minutes.
- Enfin, la gestion d'une base de données contenant les scores de plusieurs parties et de plusieurs joueurs est abordée. L'ensemble des quatre questions permet d'évaluer correctement les candidats sur l'écriture de requêtes SQL.

Ce sujet est progressif et intéressant. Le contexte du jeu de grille est séduisant, même si la référence à Tetris ne fonctionne peut-être pas aussi bien avec les élèves qu'avec leurs professeurs, qui d'ailleurs peuvent être déçus du lien finalement quelque peu éloigné avec le jeu original. Si les questions 14 et 15 ne sont accessibles qu'aux meilleurs candidats dans le temps imparti, les autres construisent un sujet bien équilibré et classant, accessible (hors partie 4) dès la première année. Les chapitres d'ingénierie numérique ne sont pas abordés.

Indications

Partie I

2 Le parcours de la grille doit se faire par deux boucles `for` imbriquées, dans le bon sens : la boucle la plus interne permet de décrire une ligne.

Partie II

3 Il faut tester chaque colonne séparément sur les `k` cases les plus élevées, mais il est inutile de tester obligatoirement toutes les colonnes. Un `return` dans une boucle `for` permet une sortie anticipée.

4 Afin de ne réaliser la modification que si la case sous le barreau existe et est vide, il est possible de placer la boucle de déplacement à l'intérieur d'un test `if`.

5 Si une des cases de destination n'est pas vide, l'algorithme doit s'arrêter sans réaliser le déplacement. L'instruction `return None` est alors toute indiquée.

6 Il faut remplacer les cases de haut en bas, en faisant très attention à traiter toutes les cases du barreau et seulement celles-là.

7 On doit commencer par parcourir la grille vers le bas à partir de `(x,y)` pour déterminer la distance de déplacement du barreau, et le déplacer alors en une seule boucle. La procédure `deplacerBarreau` n'est pas utilisable.

Partie III

9 Il faut, lors de l'unique parcours autorisé de `rangee`, compter le nombre de cases adjacentes de même couleur. À chaque changement de couleur, on peut remettre à zéro un compteur après avoir modifié `marking` sur le bon nombre de cases. Attention à ne pas oublier l'éventuel dernier alignement.

10 Utiliser la fonction `detecteAlignement` en construisant la bonne rangée avant.

11 Il faut appliquer consécutivement la fonction `scoreRangee` sur toutes les rangées pouvant exister dans la grille. Il est donc important de les prévoir correctement, en s'aidant de schémas. Attention à ne pas traiter deux fois la même rangée.

12 Plusieurs techniques sont possibles. On peut par exemple parcourir chaque colonne de bas en haut en gardant dans une variable supplémentaire la position de la case de destination des déplacements.

Partie IV

14 Pour que la fonction puisse s'exécuter récursivement sans revenir sans cesse sur les mêmes cases, il faut modifier `grille` en effaçant les cases déjà parcourues. Les codes donnés dans l'énoncé aux figures 9 et 10 ne font pas partie de cette question.

15 La principale difficulté est dans la fonction `exploreHorizontal`. Il faut essayer de comprendre son fonctionnement global et d'en déduire quel genre de disposition de cases pourrait conduire à en oublier.

Partie V

17 Il faut compter le nombre de scores meilleurs que `s`.

18 Il s'agit ici d'obtenir la valeur maximale d'un champ.

19 Sélectionner les joueurs ayant un meilleur score que le joueur concerné nécessite l'insertion d'une sous-requête dans la clause `WHERE` et une agrégation de résultats. Le comptage des joueurs doit s'effectuer dans un deuxième temps, à l'aide d'une deuxième agrégation.

I. Initialisation et affichage de l'aire de jeu

1 La fonction `creerGrille` doit créer une par une chaque case de la grille et les assembler dans les deux directions.

```
def creerGrille(largeur,hauteur):
    grille = []
    for i in range(largeur):
        colonne = []
        for j in range(hauteur):
            colonne.append(VIDE)
        grille.append(colonne)
    return grille
```

> D'après l'énoncé, `VIDE` est une variable globale, déjà définie. Il ne faut donc pas l'entourer de guillemets, ni réaliser d'affectation de cette variable.
>
> Attention à ne pas recopier `largeur` fois la même colonne. En effet, les colonnes étant des listes, créer une unique colonne et la copier plusieurs fois conduirait à créer des colonnes aux valeurs toujours identiques. Par exemple,
>
> ```
> colonne = [0,0,0]
> grille = []
> for i in range(4):
>     grille.append(colonne)
> grille[2,1] = 1
> ```
>
> construit une `grille` égale à `[[0,1,0],[0,1,0],[0,1,0],[0,1,0]]`.
>
> Créer chaque colonne en une ligne à l'aide de la syntaxe `liste*entier` permet de réaliser un code correct plus compact :
>
> ```
> def creerGrille(largeur,hauteur):
>     grille = []
>     for i in range(largeur):
>         grille.append([VIDE]*hauteur)
>     return grille
> ```
>
> L'énoncé rappelle en préambule comment définir en compréhension une liste contenant n occurrences d'une valeur k. Ce type de syntaxe permet en effet d'écrire un code bien plus compact avec
>
> ```
> def creerGrille(l,h):
>     return [ [ VIDE for j in range(h) ] for i in range(l) ]
> ```

2 La procédure `afficheGrille` doit lire chaque ligne en partant de la plus « haute », c'est-à-dire la fin de chaque colonne.

```
def afficherGrille(grille):
    largeur = len(grille)
    hauteur = len(grille[0])
    for j in range(hauteur-1,-1,-1):
        for i in range(largeur):
            if grille[i][j] == VIDE:
                afficheBlanc()
            else:
                afficheCouleur(grille[i][j])
        nouvelleLigne()
```

La procédure `nouvelleLigne` peut aussi être positionnée en début de première boucle. Les indications de l'énoncé sont insuffisantes pour choisir la meilleure position, mais celle adoptée ici correspond à une écriture classique des procédures `afficheBlanc`, `afficheCouleur` et `nouvelleLigne` de la forme

```
def afficheBlanc():
    print(' ',end='')
def nouvelleLigne():
    print()
```

où `nouvelleLigne` crée ainsi un retour à la ligne, inutile en début d'affichage mais nécessaire à la fin.

II. Création et mouvement du barreau

3 La fonction `grilleLibre` permet de déterminer si `k` cases verticalement adjacentes sont vides en haut de grille. Dès que l'on obtient un tel alignement, il est donc possible d'arrêter la recherche.

```
def grilleLibre(grille,k):
    largeur = len(grille)
    hauteur = len(grille[0])
    for i in range(largeur):
        # Marqueur indiquant si la colonne a assez de cases libres
        placelibre = True
        # Si une case est prise, on change le marqueur
        for j in range(hauteur-k, hauteur):
            if grille[i][j] != VIDE:
                placelibre = False
        # Si on a trouvé une colonne disponible, on s'arrête
        if placelibre:
            return True
    # Si on arrive ici, il n'y a aucune colonne disponible
    return False
```

Dans le pire des cas, on inspecte les `k` plus hautes cases de chaque colonne de la grille. L'inspection d'une case, contenue dans la deuxième boucle, est de complexité constante. La complexité totale est donc

$$\boxed{\mathrm{O}(k \times \texttt{largeur})}$$

L'hypothèse de non-tassement de la grille semble étonnante, la grille étant naturellement toujours tassée lorsqu'un barreau apparaît. On peut imaginer qu'il s'agit simplement de s'assurer que la seule réponse correcte soit de complexité $\mathrm{O}(k \times \texttt{largeur})$.