CORRIGÉ DU DS INFO n°2 - X-ENS 2019 -

Partie I.

Question 1. On fait attention à bien créer une nouvelle liste à chaque étape. En effet, si l'on écrit quelque chose comme :

```
largeur, hauteur = 6, 12
VIDE = [ [] ]
colonne = VIDE * hauteur
grille = [colonne] * largeur

print([id(x) for x in grille])
```

```
[2810006083464, 2810006083464, 2810006083464, 2810006083464, 2810006083464, 2810006083464]
```

vous pouvez constater que toutes les sous-listes de `grille` ont la même adresse mémoire, donc sont en fait le même objet. Ce comportement est normal, et est dû à la façon dont sont gérées les listes par `Python`.

Il faut donc écrire plutôt :

```
def creerGrille(largeur, hauteur):
    return  [[VIDE for j in range(hauteur)] for i in range(largeur)]
```

Question 2.

```
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][-j-1] # en utilisant les index négatifs
            # sinon, plus classique:   c = grille[i][hauteur-j-1]
            if c == VIDE:
                afficherBlanc()
            else:
                afficherCouleur(c)
```

Partie II.

Question 3. On teste les k dernière cases (celles du haut) pour chaque colonne. Elles ont donc pour index `hauteur-1` ... `hauteur-k`, ou bien `-1` ... `-k` si l'on utilise des indexations négatives.
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])
    for i in range(largeur):
        convient = True
        for j in range(-k, 0):
            if grille[i][j] != VIDE:
                convient = False
                break
        if convient:
            return True
    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 $\mathcal{O}(\texttt{k} \times \texttt{largeur})$.

Question 4. Si la case sous le barreau existe et est vide, on fait descendre le barreau 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]
        # ou bien plus rapidement, en utilisant le slicing:
        #  grille[x][y-1:y+k-1] = grille[x][y:y+k]
        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 contigües sont toutes 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
        convient = True
        j = 0
        # on vérifie s'il y a de la place à côté
        while j < k and convient:
            convient &= grille[x+direction][y+j] == VIDE
            # & signifie AND
            j += 1
        if convient:
            for j in range(y, y+k):
                grille[x+direction][j] = grille[x][j]
                grille[x][j] = VIDE
```

Question 6.

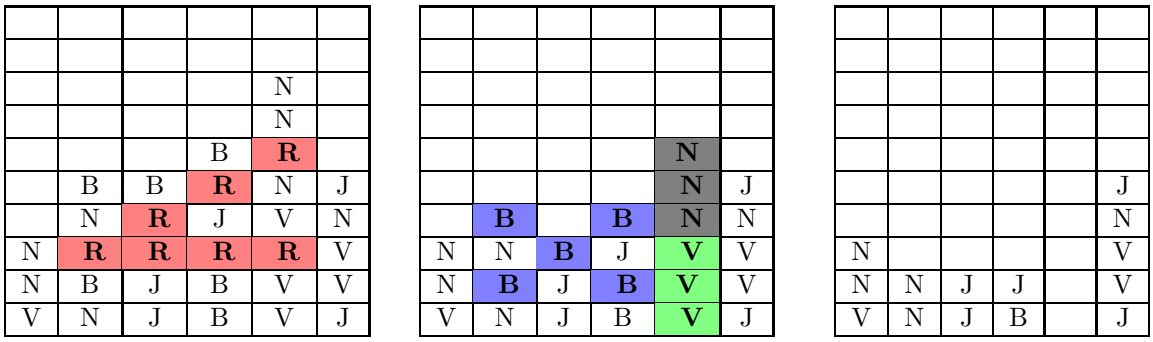
```
def permuterBarreau(grille, x, y, k):
    caseHaut = grille[x][y+k-1]
    for j in range(y+k-1, y, -1):
        grille[x][j] = grille[x][j-1]
    grille[x][y] = caseHaut
```

Question 7. Si l'on réitère la procédure `descente` tant que c'est possible, cette procédure étant de complexité $O(\texttt{k})$, on aura une complexité maximale de $O(\texttt{k} \times \texttt{hauteur})$, ce qui est clairement exclu par l'énoncé.
On cherche donc déjà de quelle valeur maximale le barreau peut descendre (c'est la variable `pas`), ce qui a une complexité en $O(\texttt{hauteur})$, puis on effectue la descente d'un seul coup, ce qui donne une complexité supplémentaire de $O(\texttt{k})$. La complexité totale est donc de $O(\texttt{hauteur+k})$.

```
def descenteRapide(grille, x, y, k):
    pas = 0
    while pas < y 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 successivement les trois configurations suivantes :



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'éventuelle dernière répétition de couleurs.

```
def detecteAlignement(rangee):
    n = len(rangee)
    marking = [False]*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 ce qui est dit dans l'énoncé.

```
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
    marking , score = 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. On parcourt ensuite toutes les rangées de la grille dans les quatre directions.

```
def effaceAlignement(grille):
    largeur = len(grille)
    hauteur = len(grille[0])
    # on fait d'abord une VRAIE copie de la grille
    g =  [[grille[i][j] for j in range(hauteur)] for i in range(largeur)]
    # on peut aussi écrire:  g = deepcopy(grille)
    # après avoir écrit dans le préambule:  from copy import deepcopy
    score = 0
    dx, dy = 1, 0 # lignes horizontales
    for j in range(hauteur):
        score +=  scoreRangee(grille, g, 0, j, dx, dy)
    dx, dy = 0, 1 # lignes verticales
    for i in range(largeur):
        score +=  scoreRangee(grille, g, i, 0, dx, dy)
    dx, dy = 1, 1 # diagonales principales
    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, -1 # diagonales secondaires
    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)
    return g, score
```

La copie de la grille a une complexité en $\mathcal{O}(\texttt{hauteur*largeur})$

La fonction `scoreRangee` si on itère en `largeur` (respectivement en `hauteur`) a une complexité en $\mathcal{O}(\texttt{hauteur})$ (respectivement en $\mathcal{O}(\texttt{largeur})$), donc chacune des boucles a une complexité en $\mathcal{O}(\texttt{hauteur*largeur})$

En conclusion la complexité de la fonction `effaceAlignement` est en $\mathcal{O}(\texttt{hauteur*largeur})$.

Question 12. Sur chaque colonne, `place` désigne la position de la première case vide et vaut `-1` tant qu'on en n'a pas rencontrée.

```
def tassementGrille(grille):
    largeur = len(grille)
    hauteur = len(grille[0])
    for i in range(largeur):
        place = - 1
        for j in range(hauteur):
            if grille[i][j] == VIDE and place == -1:
                place = j
            elif grille[i][j] != VIDE and place != -1:
                grille[i][place] = grille[i][j]
                grille[i][j] = VIDE
                place += 1
```

Question 13.

```
    def calculScore(grille):
        largeur = len(grille)
        hauteur = len(grille[0])
        g = grille # grille avant
        score, scorePartiel = 0, 1
        while scorePartiel !=0:
            g, scorePartiel = effaceAlignement(g)
            if scorePartiel != 0:
                score += scorePartiel
                tassementGrille(g)
        # on recopie (vraiment) g dans la grille initiale pour la modifier
        for i in range(largeur):
            for j in range(hauteur):
                grille[i][j] = g[i][j]
        return score
```

Partie V.

Question 16.

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

ou bien :

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

Question 17.

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

Question 18.

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

ou bien :

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

Question 19.

```
SELECT COUNT(*) +1 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);
```

ou bien

```
SELECT COUNT(*) +1 FROM
(SELECT id_j FROM PARTIES GROUP BY id_j HAVING MAX( score ) >
 (SELECT MAX( SCORE ) FROM JOUEURS JOIN PARTIES ON id_j = id_joueur WHERE nom = cc)
)
```