

X-ENS 2019 - MP - PC - PSI

Remarque : Attention, comme l'indique le rapport de jury (celui des MP) :

« L'énoncé décrivait précisément les opérations autorisées sur les listes : création de liste vide, `len`, `append`, accès via `L[i]` pour $0 \leq i < \text{len}(L)$, 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 »

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

```
Q1. def creerGrille(largeur, hauteur):  
    return [[VIDE for j in range(hauteur)] for i in range(largeur)]  
# Alternative  
def creerGrille(largeur, hauteur):  
    grille = []  
    for i in range(largeur):  
        grille.append([])  
        for j in range(hauteur):  
            grille[i].append(VIDE)  
    return grille
```

```
Q2. def afficheGrille(grille):  
    largeur=len(grille)  
    hauteur=len(grille[0])  
    for j in range(hauteur):  
        if j!=0:  
            nouvelleLigne()  
        for i in range(largeur):  
            case=grille[i][hauteur-j-1]  
            if case == VIDE:  
                afficherblanc()  
            else:  
                afficherCouleur(case)
```

Partie II. Création et mouvement du barreau

```
Q3.    largeur=len(grille)  
    hauteur=len(grille[0])  
    i=0  
    while i < largeur:  
        j=hauteur-k  
        while j<hauteur and grille[i][j]==VIDE:  
            j+=1  
        if j==hauteur:  
            return True  
        i+=1  
    return False
```

Q4. Le sujet ne précise pas ce qu'il en est de la mise à jours de `y`, dans le code jouable cette fonction renvoie `y` (de plus un teste permet de savoir que la piece a terminé de tomber et qu'il faut donc en créer une nouvelle)

```
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
```



```

    if ncoul==VIDE or ncoul!=couleur:
        # On est en fin de chaine
        if lbloc>2:
            score+=lbloc-2
            for j in range(lbloc):
                marking[i-1-j]=True
            lbloc=1
            couleur=ncoul
        else:
            # La chaine continue
            lbloc+=1
    #On n'oublie pas la dernière chaine
    if lbloc>2:
        score+=lbloc-2
        for j in range(lbloc):
            marking[n-1-j]=True
    return (marking, score)

```

Q10. Attention le sujet autorise $(dx,dy)=(0,0)$, ce qui peut provoquer une boucle infinie (mais sera évitée aux questions suivantes).

Sinon on suit ce qui est demandé dans l'énoncé et tout se passe bien. On fait attention à ne pas toucher grille

```

def scoreRangee(grille, g, i, j, dx, dy):
    if dx==0 and dy==0:
        return 0
    else:
        largeur=len(grille)
        hauteur=len(grille[0])
        x,y=i,j
        rangee=[]
        while x>=0 and x<largeur and y>=0 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

```

Q11. Il ne faut pas oublier de prendre toutes les diagonales.

Pour la complexité, copie à une complexité de $O(\text{hauteur} \times \text{largeur})$, la fonction `detectAlignement` à une complexité de $O(n)$ (où $n=\text{len}(\text{rangee})$). Ainsi `scoreRangee` a une complexité en O du nombre d'élément dans la ligne (horizontal/vertical/diagonal) qu'on considère.

Finalement, pour `effaceAlignement` on applique `scoreRangee` pour toutes les lignes horizontal (donc en $O(\text{hauteur} \times \text{largeur})$), verticales (aussi en $O(\text{hauteur} \times \text{largeur})$) et diagonales qui sont aussi en $O(\text{hauteur} \times \text{largeur})$ (sans doute mérite d'être détaillé, mais comme le rapport de jury ne râle pas sur ça ...).

In fine la complexité est en $O(\text{hauteur} \times \text{largeur})$.

```

def copie(grille):
    largeur=len(grille)
    hauteur=len(grille[0])
    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

    # Horizontal
    dx,dy=1,0
    for j in range(hauteur):
        score+= scoreRangee(grille, g, 0, j, dx, dy)

```

```

# Vertical /
dx,dy=0,1
for i in range(largeur):
    score+= scoreRangee(grille, g, i, 0, dx, dy)

# Diagonal /
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)

# Diagonal \
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)

return (g,score)

```

Q12. L'idée est de parcourir chaque colonne à partir du bas, on garde en mémoire le nombre de case vide en dessous de la case considérée et on fait descendre chaque case non vide.

```

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

```

Q13. On applique effaceAlignement() (qui fait la copie de grille) puis tassement, et on répète tant que l'opération fait augmenter le score.

```

def calculScore(grille):
    largeur=len(grille)
    hauteur=len(grille[0])
    score=0

    g,nscore=effaceAlignement(grille)
    tassementGrille(g)
    score+=nscore
    # On s'arrete quand le tassement ne fait pas augmenter le score
    while newscore !=0:
        g,nscore=effaceAlignement(g)
        tassementGrille(g)
        score+=nscore
    # On met a jours la grille
    for x in range(largeur):
        for y in range(hauteur):
            grille[x][y]=g[x][y]
    return score

```

Partie IV. Variante du jeu : régions unicolores

Q14. Le programme se termine bien, car à chaque appel récursif le nombre de case non vidée de la composante connexe de (x, y) baisse strictement.

```

def tailleRegionUnicolore(grille, x, y):
    g=copie(grille)

```

```

    return tailleRegionUnicoloreRec(g, x, y)

def tailleRegionUnicoloreRec(grille, x, y):
    largeur = len(grille)
    hauteur = len(grille[0])

    nb = 1
    couleur = grille[x][y]
    grille[x][y] = VIDE
    for dx in [-1, 1]:
        if x + dx >= 0 and x + dx < largeur and grille[x + dx][y] == couleur:
            nb += tailleRegionUnicoloreRec(grille, x + dx, y)
    for dy in [-1, 1]:
        if y + dy >= 0 and y + dy < hauteur and grille[x][y + dy] == couleur:
            nb += tailleRegionUnicoloreRec(grille, x, y + dy)
    return nb

```

- Q15. Si on prend

R	R
N	R
R	R

 et qu'on part de (0,0), le rouge tout en haut à gauche ne sera pas détecté, en effet dans exploreRegion, il n'y aura que l'exploration horizontale vers la droite, mais dans cette fonction, on ira que vers la droite (et jamais vers la gauche puisqu'il n'y a que des +dir), il ne sera donc pas détecté et la fonction ne détectera que 4 blocs dans la zone.

Partie V. Gestion du score en SQL

- Q16.

```
SELECT date,duree,score
FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
WHERE nom=cc
ORDER BY date
```
- Q17.

```
SELECT 1+COUNT(*)
FROM PARTIES
WHERE score>s
```
- Q18.

```
SELECT MAX(score)
FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
WHERE pays='France'
```
- Q19.

```
SELECT 1+ COUNT(*)
FROM ( SELECT id_j
      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
                        )
    )
```