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]` *et j'ai bien fait, vu le rapport.*

```
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 O(`hauteur`) puis la boucle inconditionnelle (`for`) qui suit a une complexité en 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` qui correspond à la couleur du bloc courant.

La variable `ncoul` correspond à la nouvelle couleur à inspecter. Si la nouvelle couleur est identique à la précédente, on incrémente la variable `lbloc`.

S'il s'agit de `VIDE`, la variable `lbloc` prend (ou garde) la valeur 1. Si `ncoul` est distincte de `couleur` 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.

Il faut par ailleurs s'efforcer de faire un seul appel à chaque valeur de la liste `rangee` ; ce qui est respecté ici.

```
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):
        ncoul=rangee[i] # nouvelle couleur à tester
        if ncoul==VIDE or ncoul!=couleur:
            if lbloc>2:
                score+=lbloc-2
                for j in range(lbloc):
                    marking[i-1-j]=True
            lbloc=1
            couleur=ncoul
        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.

Toutefois, le rapport suggère un rique de boucle infinie alors que cela ne risque pas d'arriver d'après la question suivante.

```
def scoreRangee(grille, g, i, j, dx, dy):
    if dx==0 and dy==0: return 0 #suite au rapport :(
    else:
        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. 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)

    #version plus élégante
    def tailleRegionUnicoloreV2(grille, x, y):
        g = copie(grille)
        largeur = len(g)
        hauteur = len(g[0])
        c = g[x][y]
        def auxrec(x, y):
            if not (0 <= x < largeur and 0 <= y < hauteur and g[x][y] == c):
                return 0
            g[x][y] = VIDE
            return 1 + auxrec(x-1, y) + auxrec(x+1, y) + auxrec(x, y-1) + auxrec(x, y+1)
        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  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) );
```