

Corrigé de l'Épreuve d'Informatique

Concours National Commun (CNC) – Session 2026

Filière MP

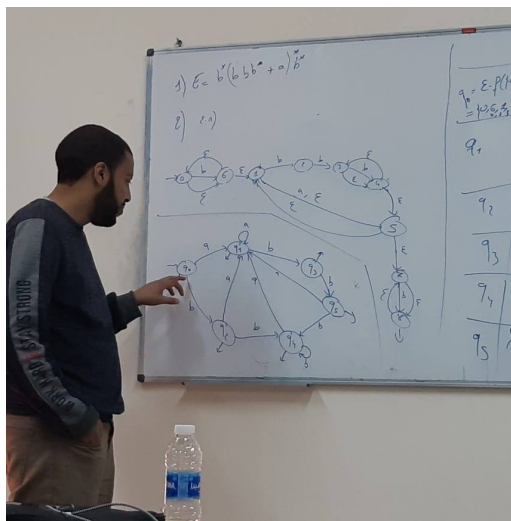
Auteur :

Yahya Kharbane

yahya.kharbane@gmail.com

Lycée d'Excellence Med VI

Casablanca



Note aux lecteurs

Ce document constitue une proposition de corrigé pour l'épreuve. Si vous repérez une erreur, une coquille, ou si vous avez la moindre question concernant les démonstrations ou le code proposés, n'hésitez pas à me contacter directement par mail !

1. Exercice : Plus grand diviseur commun

Q1.

```
def existe(L, x):  
    n = len(L)  
    for i in range(n):  
        if L[i] == x:  
            return True  
    return False
```

Q2.

Soit n la taille de la liste L . Dans le pire des cas (lorsque l'élément x n'est pas dans la liste ou se trouve à la toute fin), la boucle `for` effectue n itérations. Chaque itération réalise une comparaison en temps constant $\mathcal{O}(1)$. La complexité temporelle asymptotique est donc :

$$\mathcal{O}(1) + \sum_{i=0}^{n-1} \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(n)$$

Il s'agit d'une complexité linéaire.

Q3.

```
def communs(A, B):  
    return [x for x in A if existe(B, x)]
```

Q4.

```
def grand(L):  
    maxi = -float('inf')  
    for x in L:  
        if x > maxi:  
            maxi = x  
    return maxi
```

Q5.

```
def diviseurs(x):  
    return [i for i in range(1, x + 1) if x % i == 0]
```

Q6.

```
def premier(x):  
    return diviseurs(x) == [1, x]
```

Q7.

```
def pgcd(x, y):  
    return grand(communs(diviseurs(x), diviseurs(y)))
```

2. Partie I : Base de données et langage SQL

Q8.

```
INSERT INTO Types (id, type)
VALUES (45, 'Mangrove');
```

Q9.

```
SELECT codP, superficie
FROM Forêts
ORDER BY superficie DESC
LIMIT 3;
```

Q10.

```
SELECT COUNT(*), MIN(superficie), MAX(superficie), AVG(superficie)
FROM Forêts;
```

Q11.

En algèbre relationnelle, la requête s'écrit :

$$\Pi_{\text{nomPays}} \left(\sigma_{\text{type}='Boréale'} \left(\text{Pays} \underset{\text{codPays=codP}}{\bowtie} \text{Forêts} \underset{\text{idTyp=id}}{\bowtie} \text{Types} \right) \right)$$

Q12.

```
SELECT nomPays
FROM Pays
JOIN Forêts ON codPays = codP
JOIN Types ON idTyp = id
WHERE type = 'Boréale'
ORDER BY nomPays ASC;
```

Q13.

```
SELECT type, SUM(superficie) AS superficie_totale
FROM Forêts
JOIN Types ON idTyp = id
GROUP BY type
HAVING SUM(superficie) > 5000000
ORDER BY superficie_totale DESC;
```

Q14.

Pour calculer le pourcentage de chaque pays par rapport à la superficie totale des forêts de type 'Tropicale', on réalise une jointure avec une sous-requête regroupée qui calcule la superficie totale par type de forêt.

```
SELECT codP, (superficie / superficie_totale) * 100 AS pourcentage
FROM Forêts
JOIN (
    SELECT id, type, SUM(superficie) AS superficie_totale
    FROM Forêts
    JOIN Types ON id = idTyp
    GROUP BY id, type
) AS t_total
ON id = idTyp
WHERE type = 'Tropicale'
ORDER BY pourcentage ASC;
```

3. Partie II : Modélisation d'un feu de forêt

Q15.

```
def matrice_zeros(n, p):
    return [[0 for j in range(p)] for i in range(n)]
```

Q16.

```
def indices(R):
    return [(i, j) for i in range(len(R)) for j in range(len(R[0]))]
```

Q17.

```
def foret_aleatoire(n, p, d):
    N = int(n * p * d / 100)
    R = matrice_zeros(n, p)
    L = indices(R)
    c = 0
    while c != N:
        k = randint(0, n * p - 1)
        i, j = L[k]
        if R[i][j] == 0:
            R[i][j] = 1
            c += 1
    return R
```

Exemple d'application visuelle (Test de validation)

Afin de visualiser la répartition spatiale aléatoire des arbres au sein de la grille, on peut générer une forêt de dimension 100×100 avec une densité de 25% et afficher graphiquement la matrice obtenue à l'aide de la bibliothèque `matplotlib` :

```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Generation d'une foret de 100x100 avec 25% de densite
M = foret_aleatoire(100, 100, 25)
```

```
# Creation d'une palette de couleurs : #663C06 (Sol nu) et 'green' (Arbres)
C = ListedColormap(['#663C06', 'green'])

# Affichage graphique
plt.imshow(M, cmap=C)
plt.axis('off')
plt.show()
```

Le rendu graphique (figure 1) permet d'observer l'homogénéité de la distribution spatiale générée par l'algorithme : les cellules d'arbres vivants (pixels verts) parsèment aléatoirement le sol nu (pixels marrons) conformément aux 25% requis.

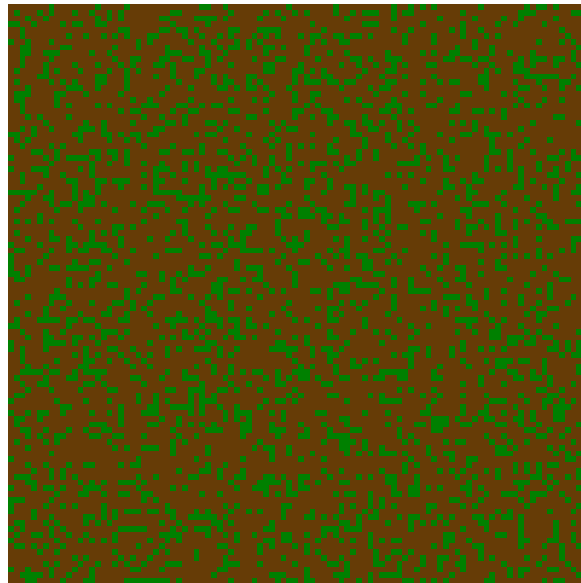


FIGURE 1 – Visualisation graphique de la grille matricielle obtenue pour $d = 25\%$

Q18.

Pour obtenir les cellules voisines sans déborder de la matrice, on utilise des vecteurs de direction. Cette méthode permet de vérifier les quatre directions (nord, est, sud, ouest) de manière élégante grâce à une boucle.

```
def voisines(R, i, j):
    n, p = len(R), len(R[0])
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    L = []
    for di, dj in directions:
        if 0 <= i + di < n and 0 <= j + dj < p:
            L.append((i + di, j + dj))
    return L
```

Q19.

Pour mettre le feu à un arbre aléatoire, on tire des coordonnées au hasard jusqu'à tomber sur une cellule contenant un arbre vivant (valeur 1).

```
def miseEnFeu(R):
    n, p = len(R), len(R[0])
    while True:
        i, j = randint(0, n - 1), randint(0, p - 1)
        if R[i][j] == 1:
            R[i][j] = 2
            return # On quitte la fonction dès que le feu est mis
```

Q20.

Pour respecter la contrainte de mise à jour simultanée sans avoir à créer une copie complète de la matrice (ce qui doublerait l'empreinte mémoire), on utilise une méthode de mise à jour en place grâce à un état temporaire (par exemple la valeur 4).

L'algorithme se déroule en trois passes :

1. On repère tous les arbres actuellement en feu (2) et on les bascule dans un état temporaire (4).
2. On parcourt la matrice à la recherche de ces états 4. Pour chacun d'eux, on met le feu (2) aux arbres vivants (1) voisins. Ces nouveaux feux ne seront pas traités lors de cette passe.
3. On transforme définitivement les états temporaires 4 en cendres (3).

```
def evolution(R):
    n, p = len(R), len(R[0])

    # Étape 1 : Marquage des arbres en feu
    for i in range(n):
        for j in range(p):
            if R[i][j] == 2:
                R[i][j] = 4

    # Étape 2 : Propagation aux voisins
    for i in range(n):
        for j in range(p):
            if R[i][j] == 4:
                L = voisines(R, i, j)
                for ip, jp in L:
                    if R[ip][jp] == 1:
                        R[ip][jp] = 2

    # Étape 3 : Passage en cendres
    for i in range(n):
        for j in range(p):
            if R[i][j] == 4:
                R[i][j] = 3
```

Q21.

Pour implémenter la boucle de propagation jusqu'à extinction complète, il est très judicieux d'introduire une fonction auxiliaire qui vérifie la présence résiduelle d'arbres en feu (état 2) dans la matrice. Cela rend la lecture de la fonction principale limpide.

```
def arbreEnFeu(R):
    n, p = len(R), len(R[0])
    for i in range(n):
        for j in range(p):
            if R[i][j] == 2:
                return True
    return False

def propagation(R):
    miseEnFeu(R)
    while arbreEnFeu(R):
        evolution(R)
```

Exemple d'application visuelle (Test de validation)

Afin de visualiser l'effet de l'incendie, on peut générer une forêt avec une densité de 65%, puis observer la matrice avant et après l'application de l'algorithme de propagation.

Il est crucial de préciser `interpolation='nearest'` dans `matplotlib` pour éviter que le lissage automatique de l'image ne crée des valeurs intermédiaires (comme la valeur 2, associée au rouge) aux frontières entre les arbres (1) et les cendres (3).

```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

M = foret_aleatoire(100, 100, 55)
# Palettes : [Terre, Arbre, Feu, Cendre]
C = ListedColormap(['#663C06', 'green', 'red', 'gray'])
C0 = ListedColormap(['#663C06', 'green'])
plt.subplot(1, 2, 1)
plt.imshow(M, cmap=C0, vmin=0, vmax=1, interpolation='nearest')
plt.axis('off')
plt.title("Avant l'incendie")
plt.subplot(1, 2, 2)
propagation(M)
plt.imshow(M, cmap=C, vmin=0, vmax=3, interpolation='nearest')
plt.axis('off')
plt.title("Après l'incendie")
plt.show()
```

La figure 2 illustre le résultat obtenu. On y observe clairement les vastes zones dévastées par l'incendie (en gris) et les quelques îlots d'arbres épargnés (en vert).

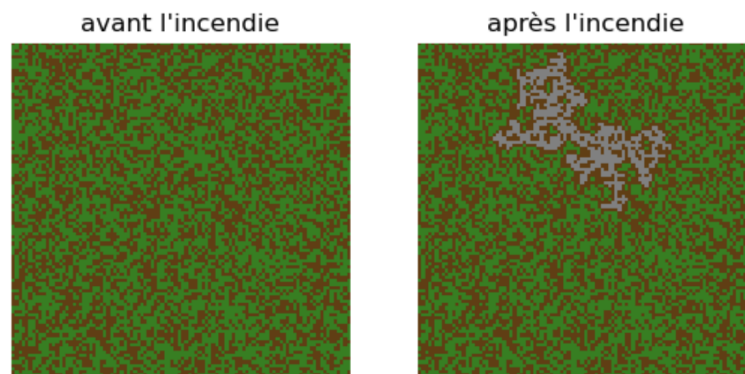


FIGURE 2 – Visualisation de la forêt avant et après propagation de l'incendie ($d = 55\%$)

Q22.

Pour calculer le taux d'arbres brûlés, on lance la propagation du feu jusqu'à son terme, puis on parcourt la grille pour faire le bilan. Une méthode très efficace consiste à compter simultanément les arbres en cendres (état 3) et les arbres survivants (état 1) lors du même parcours final. Le nombre initial d'arbres étant simplement la somme des deux, cela évite de devoir parcourir la matrice avant l'incendie.

```
def taux_brule(R):
    propagation(R)
    n, p = len(R), len(R[0])
    brule = 0
    non_brule = 0

    for i in range(n):
        for j in range(p):
            if R[i][j] == 3:
                brule += 1
            elif R[i][j] == 1:
                non_brule += 1

    total_arbres = brule + non_brule

    return 100 * brule / total_arbres
```

Q23.

```
def percolation(n, p):
    densite = [d for d in range(40, 91, 2)]
    taux_brulures = []
    for d in densite:
        R = foret_aleatoire(n, p, d)
        Tb = taux_brule(R)
        taux_brulures.append(Tb)
    plt.plot(densite, taux_brulures)
    plt.grid()
    plt.show()
```

La figure 3 illustre ce phénomène avec le tracé de la courbe générée par la fonction. On y observe très nettement le "saut" spectaculaire du taux d'arbres brûlés lorsque la densité franchit le seuil critique d'environ 60%.

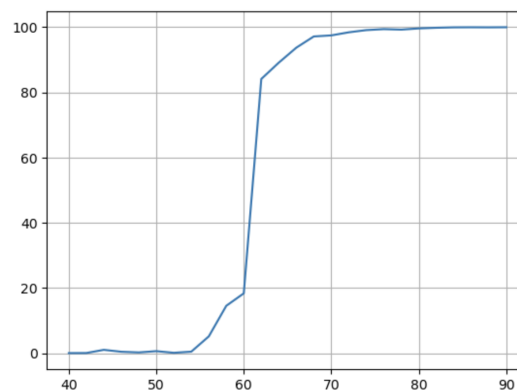


FIGURE 3 – Courbe de percolation : Taux d'arbres brûlés en fonction de la densité

4. Partie III : Détection d'objets dans une image (HOG)

Q24.

```
def som_prods(A, B):  
    s = 0  
    for i in range(3):  
        for j in range(3):  
            s += A[i, j] * B[i, j]  
    return s
```

Remarque : En exploitant pleinement *numpy*, on aurait pu simplement écrire `return np.sum(A * B)`.

Q25.

L'utilisation du *slicing* de *numpy* permet d'extraire la sous-matrice de manière très concise.

```
def voisins(M, i, j):  
    return M[i-1:i+2, j-1:j+2]
```

Q26.

```
def gradH(M):  
    n, p = M.shape # ou n, p = len(M), len(M[0])  
    X = np.zeros((n, p))  
    H = np.array([[ -1, 0, 1], [ -2, 0, 2], [ -1, 0, 1]])  
  
    for i in range(n):  
        for j in range(p):  
            if i in (0, n-1) or j in (0, p-1):  
                X[i, j] = M[i, j]  
            else:  
                X[i, j] = som_prods(voisins(M, i, j), H)  
    return X
```

Q27.

Le filtre de Sobel vertical est exactement la transposée du filtre horizontal.

```
def gradV(M):  
    n, p = M.shape  
    Y = np.zeros((n, p))  
    # On transpose la matrice de Sobel horizontale  
    H = np.array([[ -1, 0, 1], [ -2, 0, 2], [ -1, 0, 1]]).T  
  
    for i in range(n):  
        for j in range(p):  
            if i in (0, n-1) or j in (0, p-1):  
                Y[i, j] = M[i, j]  
            else:  
                Y[i, j] = som_prods(voisins(M, i, j), H)  
    return Y
```

Q28.

La fonction calcule la norme du vecteur gradient en chaque pixel. L'utilisation de `int()` permet de tronquer le résultat pour respecter la partie entière demandée par la formule mathématique.

```
def gradients(M):
    n, p = M.shape # ou n, p = len(M), len(M[0])
    G = np.zeros((n, p))
    X = gradH(M)
    Y = gradV(M)

    for i in range(n):
        for j in range(p):
            G[i, j] = int((X[i, j]**2 + Y[i, j]**2)**0.5)
    return G
```

Remarque : En exploitant pleinement la vectorisation offerte par la bibliothèque `numpy`, on aurait pu s'affranchir des boucles imbriquées et écrire ce calcul en une seule ligne très élégante : `G = np.floor(np.sqrt(X**2 + Y**2)).astype(int)`.

Exemple d'application visuelle

Afin d'illustrer l'efficacité de cet algorithme, l'application de la fonction `gradients` sur une photographie permet de réaliser une détection de contours (figure 4).

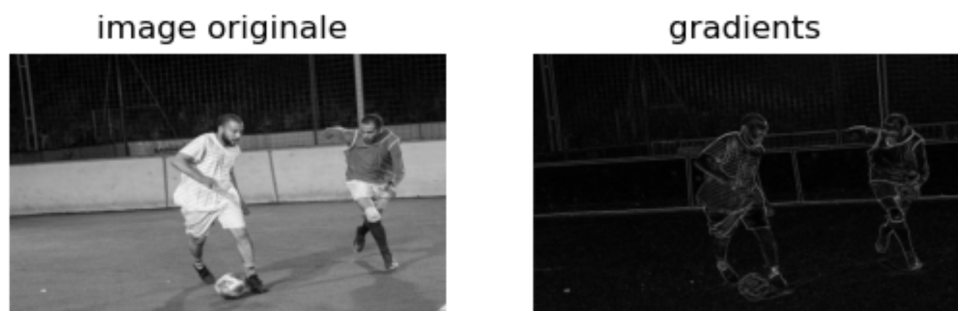


FIGURE 4 – Extraction des contours par calcul de la norme des gradients

Q29.

La fonction `atan2` renvoie un angle en radians, généralement compris dans l'intervalle $[-\pi, \pi]$. Pour obtenir une orientation en degrés strictement positive dans la plage $[0, 360[$, on effectue une conversion en degrés suivie d'une opération modulo 360.

```
def orientations(M):
    n, p = M.shape # ou n, p = len(M), len(M[0])
    D = np.zeros((n, p))
    X = gradH(M)
    Y = gradV(M)

    for i in range(n):
        for j in range(p):
            angle_rad = np.atan2(X[i, j], Y[i, j])
            D[i, j] = ((180 * angle_rad) / np.pi) % 360

    return D
```

Q30.

L'algorithme de calcul de l'histogramme d'un secteur repose sur la répartition des orientations dans 8 intervalles (ou *bins*) de 45 degrés. L'utilisation de la division entière `// 45` permet de déterminer directement l'indice de la liste `H` correspondant à l'angle, évitant ainsi une longue série de conditions `if/elif`.

```
def hist_secteur(M, i, j):
    H = [0] * 8
    G = gradients(M)
    D = orientations(M)

    for ip in range(i, i + 8):
        for jp in range(j, j + 8):
            k = int(D[ip, jp]) // 45
            H[k] += G[ip, jp]

    return H
```

Remarque sur la complexité : D'un point de vue algorithmique, la signature imposée par le sujet pour cette fonction force à recalculer `gradients(M)` et `orientations(M)` (des opérations coûteuses sur toute l'image) lors de l'appel sur chaque secteur. Dans une implémentation industrielle optimisée, on pré-calculerait `G` et `D` une seule fois de manière globale, puis on les passerait en paramètres.

Q31.

Pour construire le descripteur HOG complet, on fait glisser une fenêtre correspondant à un bloc (soit 2×2 secteurs, donc 16×16 pixels) sur toute l'image, avec un pas de 8 pixels (1 secteur).

```
def HOG(M):
    V = []
    n, p = len(M), len(M[0])

    # La borne supérieure de range est exclusive.
    # Pour que la dernière valeur de i soit (n - 16), la borne doit être > (
    # n - 16).
    for i in range(0, n - 8, 8):
        for j in range(0, p - 8, 8):
            # Concaténation des histogrammes des 4 secteurs du bloc courant
            V += hist_secteur(M, i, j)
            V += hist_secteur(M, i, j + 8)
            V += hist_secteur(M, i + 8, j)
            V += hist_secteur(M, i + 8, j + 8)

    return V
```

Remarque : Comme souligné à la question précédente, cette implémentation qui respecte strictement le découpage par fonctions demandé par l'énoncé recalculera le gradient et les orientations globales 4 fois par bloc. Dans une architecture de production, on factoriserait les appels à `gradients(M)` et `orientations(M)` tout en haut de la fonction `HOG`.

— FIN DE L'ÉPREUVE —