PC

Lycée Lakanal

2025 - 2026

Devoir d'informatique, corrigé

Partie 1

Q1. On dispose donc d'une photo de $Nl \times Nc$ pixels, chacun de ces pixels étant lui-même représenté par une liste de 3 flottants. Au départ, la photo est représentée par un tableau numpy ayant Nc lignes et Nc colonnes, et dont les coefficients sont eux-mêmes des listes de 3 flottants. Que l'on ait un tableau numpy n'a pas d'importance, on aurait pu aussi bien partir d'une image représentée par une liste de Nl sous-listes (les lignes), elles-mêmes ayant Nc coefficients, chaque coefficient étant un triplet de 3 flottants. Dans l'énoncé, $Nl = Nc = 100$.

On veut ici écrire une fonction ayant pour paramètre d'entrée une photo de ce type, et renvoyant une liste de $Nl \times Nc \times 3$ flottants, contenant les coefficients de tous les pixels de l'image. On propose par exemple :

```
def Analyse(Image):
    Nl,Nc = Image.shape[0:2]
    L_RGB = []
    for l in range(Nl):
        for c in range(Nc):
            R,G,B = Image[l,c]
            L_RGB += [R,G,B]
    return L_RGB
```

Q2. Il suffit d'écrire :

```
RGB_entr=[Analyse(Image) for Image in L_entr]
RGB_test=[Analyse(Image) for Image in L_test]
```

Si l'on préfère écrire une fonction pourquoi pas, mais dans ce cas, ne pas en écrire deux !

Par exemple, on pouvait écrire :

```
def transfo_RGG(liste):
    """
    entrée : liste = liste de photos du format décrit par l'énoncé
    sortie : liste_RGB = liste dont le ième coefficient est la liste L_RGB de
    la i ème photo contenue dans liste
    """
    liste_RGB=[Analyse(Image) for Image in liste]
    return liste_RGB
```

puis :

```
RGB_entr = transfo_RGG(L_entr)
RGB_test = transfo_RGG(L_test)
```

Q3. C'est du cours de première année

```
def maxi(L):
    indice_max, valeur_max = 0, L[0]
    for i in range(1,len(L)):
        if L[i] > valeur_max:
            indice_max, valeur_max = i,L[i]
    return valeur_max , indice_max
```

Q4. On peut utiliser une fonction annexe **compte** :

```
def compte(L):
    """ renvoie une liste Nb = liste de même longueur que L
        Nb[i] = le nombre de fois où l'élément L[i] apparaît dans L
    """
    n = len(L)
    Nb = [0]*n
    for i in range(n):
        for j in range(n):
            if L[j] == L[i]:
                Nb[i] += 1
    return Nb

def majoritaire(L):
    """ utilise les fonctions compte et maxi pour répondre à
        la question posée
    """
    Nb = compte(L)
    return maxi(compte)[0]   # maxi renvoie un couple, dont seul le premier élément
                             # nous intéresse ici.
```

Q5. Il suffit d'appliquer la formule donnant la distance euclidienne usuelle entre deux vecteurs, rappelée par l'énoncé :

```
def Distance_uv(u,v):
    n = len(u)
    Dst = 0
    for i in range(n):
        di = u[i]-v[i]
        Dst += di**2
    Dst = Dst**(1/2)
    return Dst
```

Q6. L'énoncé écrit pratiquement cette fonction, il suffit de le lire et de transcrire en python :

```
def Distance_totale(u,Lv):
    Ld = []
    for i in range(len(Lv)):
        v = Lv[i]
        Dst = Distance_uv(v,u)
        Res = [Dst,i]
        Ld.append(Res)
    return Ld
```

Q7. On a revu depuis les tris usuels au programme. On adapte par exemple ici un tri à bulle, qui, lorsque le paramètre d'entrée est une liste de longueur n, a une complexité en $O\left(n^2\right)$:

```
def Tri(L):
    n=len(L)
    for i in range(n):
        for j in range(n-1):
            if L[j][0]<L[j+1][0]:
                L[j],L[j+1]=L[j+1],L[j]
```

Q8. Il s'agit de commencer à assembler les fonctions précédentes : on applique la fonction **distance_totale** de la question **Q6**, on trie la liste qu'elle renvoie, et il ne reste plus qu'à retenir les k éléments de la liste ainsi obtenue :

```
def Proches(u,Lv,k):
    Ld = Distance_totale(u,Lv)
    Tri(Ld)
    Res = Ld[:k]
    return Res
```

Q9. On continue l'assemblage : on applique la fonction précédente, on obtient ainsi les k plus proches voisins de u issus de la liste **RGB_entr** . On se sert ensuite de la liste **Num_entr** pour déterminer quel type de panneau représente chacun de ces voisins, et enfin, à l'aide de la fonction **majoritaire** de **Q4** , on détermine quel est le type de panneau qui est majoritaire parmi ces k plus proches voisins de u .

```
def decision(u,k):
    Voisins = Proches(u,RGB_entr,k)
    Types_voisins = [Num_entr[voisin[1]] for voisin in Voisins]
    return majoritaire(Types_voisins)
```

Q10. On applique la fonction **decision** précédente à tous les vecteurs éléments de la liste **RGB_test** :

```
Num_test_KNN = [decision(u,k) for u in RGB_test]
```

Q11. Il y avait ici une incohérence dans l'énoncé : si $p = 4$, les types de panneaux sont numérotés de 0 à 3 … deux possibilités pour gommer cette erreur d'indice : supposer que $p = 5$, ou bien ignorer les 4 dans les listes données par l'énoncé.

Si l'on ignore les 4 figurant dans les listes données :

On obtient pour matrice de confusion : $M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

Si l'on suppose que p = 5 :

On obtient pour matrice de confusion : $M = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix}$.

Q12. On propose la fonction suivante :

```
def confusion(p,L1,L2):
    M = np.zeros((p,p))
    n = len (L1)
    for k in range(n):
        j=L1[k]  #type de panneaux réel
        i=L2[k]  #type de panneaux diagnostiqué
        M[i][j] + = 1
    return M
```

Q13. On écrit :

```
def taux(M):
    p = len(M[0])
    liste_taux = []
    for i in range(p):
        nb_i = sum(M[:][i])  #nombre de panneaux réellement de type i
        pos_i = M[i][i]*100/nb_i

        nb_pas_i = 0          #nombre de panneaux réellement non de type i
        nb_pas_i_pas_diagnostique_i = 0  #nombre de panneaux réellement non de type i, et non diagnostiqués i
        for a in range(p)  :
            for b in range(p) :
                if b != i :
                    nb_pas_i += M[a][b]
                    if a != i:
                        nb_pas_i_pas_diagnostique_i += M[a][b]
        neg_i = nb_pas_i_pas_diagnostique_i*100/nb_pas_i
        liste_taux.append((pos_i,neg_i))
    return liste_taux
```

Partie 2

Q.14. L'énoncé est légèrement ambigu : un "vecteur" est-il représenté ici comme une liste, ou comme un array ?

A priori, un vecteur, ce serait plutôt un array…

Voici une fonction qui est valable dans les deux cas, mais plutôt orientée vers une représentation par liste :

```
def barycentre(L):
    """ entrée : L = liste de vecteurs
                     L[k] est une liste de taille n représentant un vecteur
        sortie : bary = liste de longueur n représentant un vecteur
    """
    n = len(L[0])
    bary = [0] * n
    for k in range(len(L)):
        point=L[k]
        for i in range(n):
            bary[i] += point[i]
    return [x/len(L) for x in bary]
```

voici maintenant une fonction valable uniquement pour une représentation par array (deux array u, v pouvant être additionnés via l'opération u + v ; alors que pour deux listes, + fait de la concaténation) :

```
def barycentre(L):
    """ entrée : L = liste de vecteurs
                     L[k] est un array numpy de taille n représentant un vecteur
        sortie : bary = array numpy de taille n représentant un vecteur
    """
    n = len(L[0])
    bary = np.zeros(n)
    for k in range(len(L)):
        bary = bary + L[k]
    return bary/len(L)
```

Q15. On suit ici l'énoncé, et on utilise la fonction **Distance_uv(u,v)** . Bien entendu, d'autres variantes étaient possibles, et notamment on aurait également pu utiliser la fonction **Distance_totale(u, Lv)** .

```
def PlusProches(u , S) :
    """ entrée : u = liste
                   = un vecteur
                 S = liste de listes
                   = liste de vecteurs
        sortie : proche = entier
                        = indice du vecteur de S qui est le plus proche
                          du vecteur u
    """
    p = len(S)
    proche = 0
    distance_proche = Distance_uv(u,S[0])
    for i in range(1,p):
        if Distance_uv(u,S[i]) < distance_proche :
            proche = i
            distance_proche = Distance_uv(u,S[i])
    return proche
```

Q16. On applique la fonction **PlusProches** à tous les vecteurs de U :

```
def repartition(U , S) :
    """ entrées et sorties décrites par l'énoncé
    """
    N = len(U)
    P = []
    for k in range(N) :
        P.append(PlusProches(U[k],S))
    return P
```

Q17. On propose :

```
def nouveaux_barycentres( p, U , P ) :
    N = len(U)
    S = []
    for i in range(p):
        L = [U[j] for j in range(N) if P[j] == i]
        # L = liste des vecteurs de ceux des vecteurs U[j]
        # de U qui sont tels que P[j] = i
        S.append(barycentre(L))
    return S
```

Q18. On suit l'algorithme tel qu'il est décrit en début de partie :

```
def pMoyennes(U, C, p) :
    P = repartition(U,C,p)              #Phase d'affectation
    S = nouveaux_barycentres(p,U,P)     #Mise à jour des barycentres
    while S != C :
        C = S
        P = repartition(U,C)            #Phase d'affectation
        S = nouveaux_barycentres(p,U,P) #Mise à jour des barycentres
    return P
```

zui

Q19. Il suffit d'appliquer la fonction **nouveaux_barycentre** aux objets adaptés :

```
def Initialisation_barycentres(Num_entr, RGB_entr , p ) :
    return nouveaux_barycentres( p, RGB_entr, Num_entr)
```

Q20. Cette suite d'instructions permet de répartir les panneaux de la liste RGB_test suivant l'algorithme des k moyennes, et non plus (comme en partie **I**) suivant l'algorithme des k plus proches voisins.

En résumé :

- des panneaux modèles, on ne retient que les barycentres (de la liste des panneaux entr ayant le type i, on ne retient que le barycentre).
- On se base sur ces p barycentres initiaux pour répartir les panneaux de la liste test en p groupes, et l'on espère naturellement au final que ce sera efficace (chaque panneau de la liste test arrivant dans le groupe qui correspond au type de panneau qu'il représente).

  Le problème est que, d'une étape à l'autre, les barycentres sont modifiés. S'il y a une erreur à l'une de ces étapes (un panneau rangé dans la mauvaise catégorie), elle est répercutée dans les étapes suivantes.

Il ne paraît pas très réaliste d'espérer que l'algorithme des k moyennes donnera un résultat plus fiable que celui des k plus proches voisins (les principes de ces deux algorithmes sont proches, mais, dans l'algorithme des k plus proches voisins, on dispose de plus de données au départ). On pourrait en revanche envisager cette utilisation de l'algorithme des k moyennes à titre de contrôle de la fiabilité des résultats obtenus en partie **I** .

Partie 3

Q21. num entreprise est une clé étrangère de la table panneaux. Expliquer.

On rappelle que :

- Une **clé primaire** d'une table est une donnée permettant d'identifier de manière unique chaque enregistrement (c'est- à-dire chaque ligne) de cette table. Ici, num est une clé primaire de la table entreprise, car chaque entreprise possède un numéro (num), qui correspond à une et une seule ligne de la table entreprise.
- Une **clé étrangère** dans une table1 est une donnée qui permet de la relier à une table2, en se référant à une clé primaire de cette table2 : ceci permet d'effectuer des jointures efficaces entre ces deux tables. Ici, **num entreprise** est une clé étrangère de la table **panneaux**, car elle renvoie à la clé primaire **num** de la table **entreprise**.

Q22. Ecrire une requête permettant de déterminer combien de panneaux de type "AB " sont présents dans la table panneaux.

```
SELECT COUNT() FROM panneaux WHERE type = "AB "
```

Rappel : il est inutile de mettre quelque chose entre les parenthèses du COUNT (sauf cas qui n'apparaîtront pas aux concours).

Q23. Ecrire une requête permettant de déterminer quelle est l'efficacité moyenne d'un panneau de type "AB ".

```
SELECT AVG(efficacite) FROM panneaux WHERE type = "AB "
```

Cette requête utilise la fonction d'agrégation **AVG** (moyenne), qui est à connaître.

Q24. Quel est le type de panneaux le plus représenté dans la table panneaux ?

Attention, le sujet ne présente que des extraits des tables, et faire des calculs sur les quelques lignes qu'il montre de chacune n'a strictement aucun intérêt : il s'agit bien sûr de répondre à l'aide de requêtes SQL !

Voici une requête répondant à la question :

```
SELECT Type, COUNT() AS Nb
  FROM Panneaux GROUP BY type
  HAVING Nb = ( SELECT MAX(COUNT()) FROM Panneaux GROUP BY type )
```

Des réponses du type :

```
SELECT Type, COUNT()
  FROM Panneaux GROUP BY type
  ORDER BY COUNT() DESC LIMIT 1
```

sont possibles et rapportent la grande majorité des points ; mais pas tous, car elles ne renvoient qu'une réponse partielle, dans le cas où il y a des égalités.

Q25. On utilise une requête imbriquée. Par ailleurs, on rappelle la fonction d'agrégation AVG (moyenne).

```
SELECT Type
   FROM Panneaux GROUP By type
   HAVING AVG(efficacite)
    = (SELECT MIN(C) FROM (SELECT AVG(efficacité)) AS C FROM Panneaux GROUP BY Type) )
```

Q26. Une jointure est, sinon absolument nécessaire, en tous cas nettement adaptée à la situation.

Le GROUPY BY ci-dessous permet d'éviter d'avoir plusieurs fois le nom de chaque entreprise concernée.

```
SELECT nom
  FROM entreprises JOIN panneaux
  ON num_entreprises = entreprises.num
  WHERE Type = 'AB'
  GROUP BY entreprises.num
```

Q27. Une requête assez complexe pour terminer… sauf très possible erreur, la requête suivante répond à la question posée :

```
SELECT nom
  FROM entreprises JOIN panneaux
  ON num_entreprises = entreprises.num
  WHERE Type = 'AB'
  GROUP BY entreprises.num
  HAVING AVG(efficacité) =
    (SELECT MIN(C) FROM
                  (SELECT AVG(efficacité) AS C FROM entreprises JOIN panneaux
                    ON num_entreprises = entreprises.num
                    WHERE Type = 'AB'
                    GROUP BY entreprises.num)
     )
```

Partie 4

Q28. $T = [1, 1.5, 2]$ signifie que le skieur 0 a pour taille 1, le skieur 1 a pour taille 1.5 et le skieur 2 a pour taille 2.

$\sigma = [1, 3, 5]$ signifie que le skieur 0 se voit attribuer la paire de skis d'indice 1, soit une paire de longueur 0.83 ; que le skieur 1 se voit attribuer la paire de skis d'indice 3, qui a pour longueur 1.75 ; et enfin que le skieur 2 récupère la paire de ski d'indice 5, qui a pour longueur 2.05.

On a donc ici : $\rho(\sigma) = |1 - 0.83| + |1.5 - 1.75| + |2 - 2.05| = 0.47$.

Q29. On applique la formule définissant $\rho(\sigma)$ dans l'énoncé :

```
def rho(T,S, Sigma) :
    """ entrées :     T = liste de flottants de longueur N
                        = liste des tailles des N skieurs
                      S = liste de flottants de longueur M
                        = liste des longueurs des M paires de skis
                  Sigma = liste d'entiers de longueur N
                          Sigma[i] = numéro de la paire de skis attribuée au skieur i
    """
    N = len(T)
    Total = 0  #somme des valeurs absolues définissant rho(sigma) dans l'énoncé
    for i in range(N) :
        Total += abs( T[i] - S(sigma[i]) )
    return Total
```

Q30. L' « algorithme suggéré » signale simplement que, lorsque les skieurs sont rangés par ordre de taille croissante, on peut trouver une distribution optimale dans laquelle les paires de skis $s_{i_0}, ..., s_{i_{n-1}}$ qui leurs sont attribuées sont elles aussi de taille croissante, ie. vérifient : $s_{i_0} \leq s_{i_1} \leq ... \leq s_{i_{n-1}}$ (quitte à échanger des skieurs de même taille). C'est évident, mais en voici une preuve formelle :

Supposons un instant qu'il n'existe pas de distribution optimale vérifiant la condition de taille croissante des skis.

Soit $\sigma \rightarrow s_{i_0}, ..., s_{i_{n-1}}$ une distribution optimale : la quantité $\rho(\sigma) = \sum_{k=0}^{n-1} \left| t_k - s_{i_k} \right|$ est donc minimale

D'un autre côté, cette distribution ne vérifie pas la condition de taille croissante des skis, il existe deux indices $0 \leq u < v \leq n-1$ tels que $s_{i_u} > s_{i_v}$. Considérons la distribution où l'on échange les paires de skis des skieurs u et v, les autres paires étant inchangées. On obtient ainsi une distribution $\sigma' \rightarrow s'_{i_0}, ..., s'_{i_{n-1}}$ avec $s'_{i_u} = s_{i_v}$, $s'_{i_v} = s_{i_u}$ et pour tout k différent de u et v, $s'_{i_k} = s_{i_k}$. On a pour cette distribution :

$$\begin{aligned}
\rho(\sigma') &= \sum_{k=0}^{n-1} \left| t_k - s'_{i_k} \right| \\
&= \sum_{\substack{k=0 \\ k \neq u, k \neq v}}^{n-1} \left| t_k - s'_{i_k} \right| + \left| t_u - s'_{i_u} \right| + \left| t_v - s'_{i_v} \right| \\
&= \sum_{\substack{k=0 \\ k \neq u, k \neq v}}^{n-1} \left| t_k - s_{i_k} \right| + \left| t_u - s_{i_v} \right| + \left| t_v - s_{i_u} \right| \\
&= \sum_{\substack{k=0 \\ k \neq u, k \neq v}}^{n-1} \left| t_k - s_{i_k} \right| + \left| t_u - s_{i_u} \right| + \left| t_v - s_{i_v} \right| + \left(\left| t_u - s_{i_v} \right| + \left| t_v - s_{i_u} \right| - \left| t_u - s_{i_u} \right| - \left| t_v - s_{i_v} \right| \right) \\
&= \rho(\sigma) + \left(\left| t_u - s_{i_v} \right| + \left| t_v - s_{i_u} \right| - \left| t_u - s_{i_u} \right| - \left| t_v - s_{i_v} \right| \right).
\end{aligned}$$

On distingue alors les cas suivants :

Si $t_u \leq t_v \leq s_{i_u} < s_{i_v}$: alors

$$\left(\left| t_u - s_{i_v} \right| + \left| t_v - s_{i_u} \right| - \left| t_u - s_{i_u} \right| - \left| t_v - s_{i_v} \right| \right) = s_{i_v} - t_u + s_{i_u} - t_v - \left(s_{i_u} - t_u \right) - \left(s_{i_v} - t_v \right) = 0,$$

donc $\rho(\sigma') = \rho(\sigma)$, et σ' est elle aussi optimale.

Si $t_u \leq s_{i_u} \leq t_v \leq s_{i_v}$:

$\left(\left| t_u - s_{i_v} \right| + \left| t_v - s_{i_u} \right| - \left| t_u - s_{i_u} \right| - \left| t_v - s_{i_v} \right| \right) = = -2 s_{i_u} + 2 t_v \geq 0$, donc $\rho(\sigma') \geq \rho(\sigma)$

On vérifie de même que si $t_u \leq s_{i_u} \leq s_{i_v} \leq t_v$, si $s_{i_u} \leq t_u \leq s_{i_v} \leq t_v$, si $s_{i_u} \leq t_u \leq t_v \leq s_{i_v}$ ou si $s_{i_u} \leq s_{i_v} \leq t_u \leq t_v$, on a également $\rho(\sigma') \geq \rho(\sigma)$.

Dans tous les cas, σ' est elle aussi optimale. On obtient donc une distribution optimale en partant de σ, et en permutant les paires de skis qui ne sont pas rangées dans l'ordre croissant : d'où une contradiction, et la fin de ce raisonnement par l'absurde.

Q31. S'il n'y a pas de skieur, il n'y a pas de problème ! $p(0, m) = 0$ pour tout m.

S'il n'y a pas de ski, d'après l'énoncé, c'est comme si on attribuait à tous les skieurs des skis de longueur 0 :

Pour tout n, $p(n, 0) = \sum_{i=0}^{n-1} \left| t_i - 0 \right| = \sum_{i=0}^{n-1} t_i$.

Q32. On a $p(n,m) = \min\left\{ p(n-1,m-1) + \left|t_n - s_m\right|,\ p(n,m-1)\right\}$:

Lorsque l'on considère le meilleur appariement des paires de skis $1,\ldots,m$ et des skieurs $1,\ldots,n$, il n'y a que deux choix possibles pour la paire de skis m :

- Soit elle n'est associée à aucun skieur et dans ce cas, il reste les paires $1,\ldots,m-1$ à attribuer aux skieurs $1,\ldots,n$: $p(n,m) = p(n,m-1)$.
- Soit elle est attribuée au skieur n (et à aucun autre : $p(n,m)$ comprend le terme $\left|t_n - s_m\right|$, et il reste à attribuer aux skieurs $1,\ldots,n-1$ des skis pami les paires $1,\ldots,m-1$). On a donc $p(n,m) = p(n-1,m-1) + \left|t_n - s_m\right|$.

Entre ces deux choix, on fait celui qui est le meilleur, d'où au final la formule annoncée :

$$p(n,m) = \min\left\{ p(n-1,m-1) + \left|t_n - s_m\right|,\ p(n,m-1)\right\}$$

Q33. On écrit une fonction récursive, les cas terminaux étant décrits en **Q31**, et le cas général en **Q32** :

```
def choix_ski(T,S):
    if len(T) == 0 :
        return 0
    elif len(S) == 0 :
        return sum(T)
    else :
        return min( choix_ski(T[:-1],S[:-1]) + abs(T[-1] - S[-1]) , choix_ski(T,S[:-1]) )
```

Q34. Dans la formule $p(n,m) = \min\left\{ p(n-1,m-1) + \left|t_n - s_m\right|,\ p(n,m-1)\right\}$:

A chaque fois que $p(n,m) < p(n,m-1)$, il faut attribuer la paire de skis m au skieur n.

Il suffit de modifier la fonction précédente de manière à garder cela en mémoire :

```
def choix_ski_memoire(T,S,U=[]):
    """   Nouveau paramètre U = liste des tailles de skis attribuées à chaque skieur
    """
    if len(T) == 0 :
        return 0, []
    elif len(S) == 0 :
        return sum(T), [0]*len(T)
    else :
        a = choix_ski(T[:-1],S[:-1])[0] + abs(T[-1] - S[-1])
        b = choix_ski(T,S[:-1])[0]
        if a < b :
            return a, choix_ski(T[:-1],S[:-1])[1] + [S[-1]]
        else:
            return choix_ski(T,S[:-1])
```

FIN