CCINP PC-PSI — 2023

Reconnaissance optique de caractères

Partie I. — Acquisition d'un document

Q 1. Pour représenter un entier entre 0 et $255 = 2^8 - 1$, il faut 8 bits, ce qui représente un octet.

La taille de l'image en pouces carrés est $S = \dfrac{21 \times 29,7}{2,5^2}$, qu'il faut multiplier par 300^2 pour obenir le nombre de pixels. On trouve $8,98.10^6$ pixels, d'où $2,16.10^8$ bits (ou encore 26,9 Mo). En prenant pour un pouce la valeur plus précise de 2,54 cm, on obtient $2,09.10^8$ bits, ou 26,1 Mo.

Q 2. Il y a une erreur dans le code de la fonction `conversion_gris` dont la ligne 3 devrait être `img = initialise(n0, n1, 0)`. Pour en évaluer la complexité, on aimerait disposer de celle des fonctions `dimension` et `initialise` utilisées dans la fonction...

Toutefois, il n'est pas raisonnable de penser qu'elles ne soient pas de complexité au plus égale à $\mathcal{O}(n_0 n_1)$. Le travail effectué à l'intérieur de la double boucle est de complexité constante (lecture d'un élément, affectation, calcul arithmétique et nouvelle affectation). Compte non tenu des deux fonctions appelées en début de code, la complexité est ainsi en $\mathcal{O}(n_0 n_1)$, d'où une complexité globale en $\mathcal{O}(n_0 n_1)$, soit linéaire en le nombre de pixels de l'image.

Q 3. On s'inspire de la fonction `conversion_gris`.

```
def binarisation(imG: array, seuil: int) -> array:
    n0, n1 = dimension(imgC)[:2]
    img = initialise(n0, n1, 0)
    for i in range(n0):
        for j in range(n1):
            if imG[i][j] > seuil:
                img[i][j] = 255
    return img
```

Partie II. — Reconnaissance du document

II.1. Rotation de l'image

La Figure 2 représente une rotation d'angle α, dont parle bien le texte, mais la matrice donnée est celle de la rotation d'angle $-\alpha$... qui est celle que l'on va utiliser (page 5). Il aurait fallu écrire

$$\begin{pmatrix} i - p//2 \\ j - q//2 \end{pmatrix} = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} n_i - p//2 \\ n_j - q//2 \end{pmatrix}.$$

Q 4. La proposition décrite dans l'énoncé est la deuxième.

```
def bilineaire (im: array, x: float, y: float ) -> int:
    x0, y0 = int(x), int(y)
    x1, y1 = x0 + 1,  y0 + 1
    a = lineaire(y, y0, y1, im[x0][y0], im[x0][y1])
    b = lineaire(y, y0, y1, im[x1][y0], im[x1][y1])
    c = lineaire (x, x0, x1, a, b)
    return int(c)
```

Q 5. On recolle les morceaux. La fonction se complète bien.

```
def rotation(im :array, angle: float ) -> array:
    p, q, _ = dimension(im)
    imr = initialise(p, q, 255)
    angr = - angle * m.pi / 180
    matR = [[m.cos(angr), - m.sin(angr)], [m.sin(angr), m.cos(angr)]]
    for ni in range(p):
        for nj in range(q):
            x, y = prod_matrice_vecteur(MatR, [ni - p//2, nj - q//2])
            x, y = x + p//2, y + q//2
            if 0 <= int(x) < p - 1 and 0 <= int(y) < q - 1: # sinon, on peut sortir de l'image
                imr[ni][nj] = bilineaire(im, x, y)
    return imr
```

Q 6. L'intérêt de coder les entiers sur 8 bits au lieu de 64 est un gain de place, l'image prenant ainsi huit fois moins de place en mémoire. Tant qu'à faire, une fois la binarisation effectuée, on pourrait aussi se contenter d'un seul bit... Sur les entiers l'opération 18 - 23 donne évidemment -5. Sur 8 bits, l'opération se fait *modulo* 256, et donne donc 251. Les opérations réalisées sur l'image par `bilineaire` donnent donc des résultats non pertinents dès que l'on sort de l'intervalle entier $[\![0, 255]\!]$.

II.2. Segmentation

Q 7. On compte les pixels noirs. On peut parcourir l'image par indice des pixels ou utiliser le parcours de liste par éléments.

```
def histo_lignes_indices(im: array) -> list:
    n, p, _ = dimension(img)
    Lnoir = []
    for i in range(n):
        compteur = 0
        for j in range(p):
            if im[i][j] == 0:
                compteur += 1
        Lnoir.append(compteur)
    return Lnoir
```

```
def histo_lignes(im: array) -> list:
    Lnoir = []
    for ligne in img:
        compteur = 0
        for pixel in ligne:
            if pixel == 0:
                compteur += 1
        Lnoir.append(compteur)
    return Lnoir
```

Q 8. Cette question est délicate. Il y a différentes manières de procéder et l'énoncé impose la sienne. Je ne vois pas comment respecter exactement ce qui est proposé si l'on veut traiter le cas où la dernière ligne contient des pixels noirs. Voici deux versions. La première suit l'énoncé en le corrigeant. La deuxième introduit une deuxième boucle correspondant mieux à ce que l'on fait si l'on dresse la liste à la main. À noter que la boucle conditionnelle `while` de l'énoncé aurait pu être remplacé par une boucle inconditionnelle.

```
def detecter_lignes(liste: list) -> list:
    lignes, i, deb, n = [], 0, -1, len(liste)
    while i < n:
        if liste[i] != 0 and deb == -1:
            deb = i
        elif liste[i] == 0  and deb != -1:
            lignes.append([deb, i - 1])
            deb = -1
        if i == n - 1 and liste[i] != 0:
            lignes.append([deb, n-1])
        i += 1
    return lignes
```

```
def detecter_lignes2(liste: list) -> list:
    lignes, i, deb, n = [], 0, -1, len(liste)
    while i < n:
        if liste[i] == 0:
            deb, i = deb + 1, i + 1
        else:
            j = i
            while j < n and liste[j] != 0:
                j += 1
            lignes.append([deb + 1, j-1])
            i, deb = j + 1, j
    return lignes
```

Pour tester les fonctions, on peut faire appel par exemple à

```
LL = [[0,0,0,1,0,1,1,1,0,0,1,1,0], [0,0,0], [1,1,1], [1,0,0,1,1], [0,0,1]]
```

Q 9. La méthode utilisée est la *méthode dichotomique*. L'hypothèse faite sur l'absence d'extremum local est nécessaire pour faire tourner l'algorithme ; il ne semble pas évident qu'elle soit vérifiée en pratique. On peut bien sûr se tourner vers une recherche de maximum classique en complexité linéaire au lieu de logarithmique.

Pour répondre à la question, l'algorithme se termine car la longueur de l'intervalle de recherche est divisée par deux à chaque étape. Au bout de n étapes, l'intervalle sera ainsi de longueur $\frac{b-a}{2^n}$, soit $\left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil$ étapes.

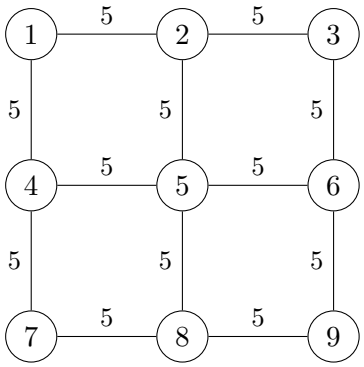
Q 10. Pas de difficulté dans cette question, qui consiste à programmer une dichotomie continue.

```
def nb_zeros(im: array, angle: float) -> int:
    imr = rotation(im, angle)
    ligne = histo_ligne(imr)
    f = ligne.count(0)
    return f

def rotation_auto(im: array, a: float, b: float) -> array:
    c = (a + b)/2
    fc = nb_zeros(im, c)
    while b - a > 0.1:
        ac, cb = (a + c)/2, (c + b)/2
        fac, fcb = nb_zeros(im, ac), nb_zeros(im, cb)
        maxi = max(fac, fc, fcb)
    if fac == maxi:
        b = c
        c = ac
        fc = fac
    elif fc == maxi:
        a, b = ac, cb
    else:
        a = c
        c = cb
        fc = fcb
    return rotation(im, (a+b)/2)
```

II.3. Restauration d'image

Q 11. Les sommets du graphe sont les neuf pixels et les arêtes, toutes de poids 5, relient les pixels adjacents (c'est-à-dire ceux situés immédiatement au-dessus, en dessous, à droite ou à gauche).



Q 12. En notant 0 pour les arêtes inexistantes, et en comprenant que le « graphe complet » dont parle l'énoncé est la totalité du graphe à neuf sommets et non un *graphe complet* au sens de la théorie des graphes, on obtient le tableau suivant.

	S	1	2	3	4	5	6	7	8	9	P
S	0	0	210	190	20	100	200	10	5	255	0
1	-	0	5	0	5	0	0	0	0	0	255
2	-	-	0	5	0	5	0	0	0	0	45
3	-	-	-	0	0	0	5	0	0	0	65
4	-	-	-	-	0	5	0	5	0	0	235
5	-	-	-	-	-	0	5	0	5	0	155
6	-	-	-	-	-	-	0	0	0	5	55
7	-	-	-	-	-	-	-	0	5	0	245
8	-	-	-	-	-	-	-	-	0	5	250
9	-	-	-	-	-	-	-	-	-	0	0
P	-	-	-	-	-	-	-	-	-	-	0

Q 13. La description de l'algorithme est longue et, malheureusement, totalement incompréhensible sur un point crucial. La technique d'augmentation du flot pour obtenir un flot maximal est connue sous le nom de *méthode de Ford-Fulkerson*. L'algorithme de Edmonds-Karp décrit dans l'énoncé reprend cette méthode en précisant de quelle manière on choisit le chemin à augmenter. En fait, la méthode de Ford-Fulkerson marche de quelque façon que l'on s'y prenne, mais le choix du chemin influe sur sa complexité ; l'idée d'Edmonds-Karp est de prendre un chemin de la source au puits le plus court possible (au sens du nombre d'arêtes qu'il contient).

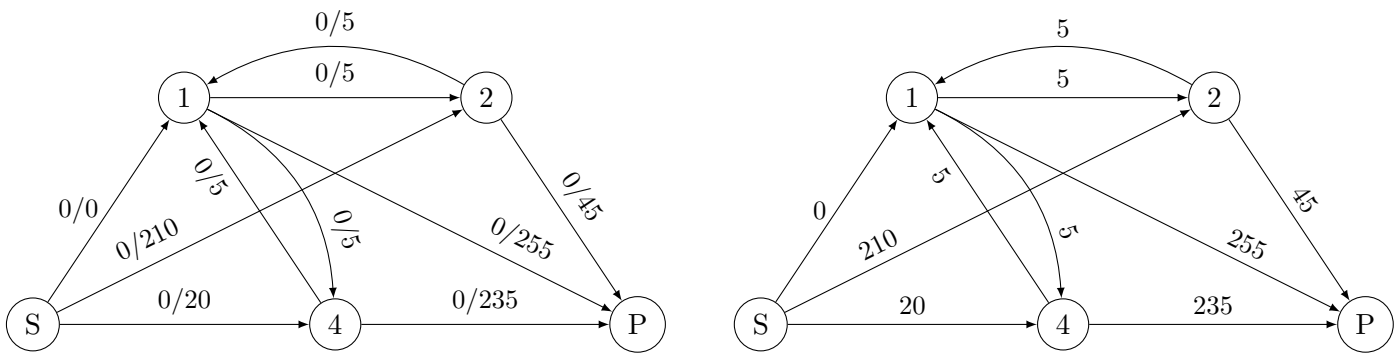
L'énoncé dit que l'on « cherche itérativement un plus court chemin C (c'est-à-dire un chemin où la somme des étiquettes du graphe résiduel en parcourant les arêtes le constituant est minimal et comportant le moins d'arêtes) de la source au puits sur lequel il n'y a pas d'arête saturée (c'est-à-dire un chemin pour lequel aucune des arêtes correspondantes du graphe résiduel n'est pondérée par 0) ». Il faut comprendre que l'on considère les chemins les plus courts non saturés (c'est l'algorithme d'Edmonds-Karp tel qu'il est décrit dans la littérature). Parmi ces chemins, l'énoncé demande de choisir celui dont la somme des étiquettes du graphe résiduel est minimal, ce qui conduit à deux solutions possibles pour le premier choix. Dans ce corrigé, on a opté pour le chemin de capacité résiduelle maximale, c'est-à-dire celui qui permet d'augmenter le flot de la plus grande valeur possible. Comment mentionné ci-dessus, cela ne modifie pas le résultat final.

Dans l'exemple proposé, les chemins de la source au puits parmi lesquels il faut choisir à chaque étape sont :

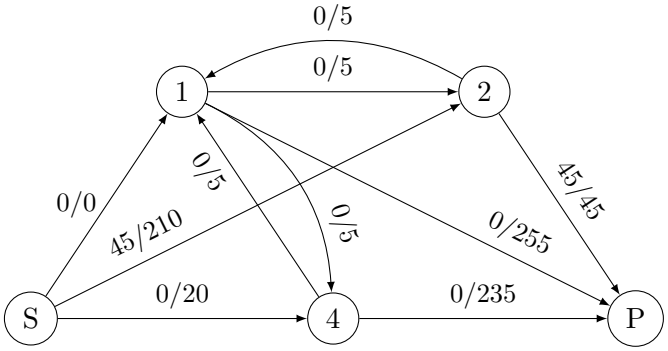
— les chemins dont la première arête est `S-1`. On ne les choisira jamais car cette arête a une capacité nulle ;

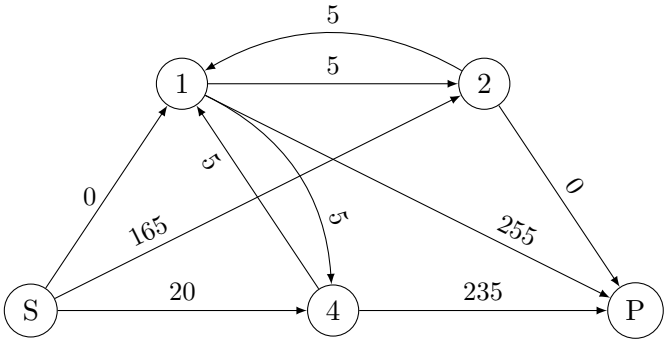
— les chemins dont la première arête est `S-2`. Par ordre croissant de longueur, ce sont `S-2-P`, `S-2-1-P` et `S-2-1-4-P` ;

— les chemins dont la première arête est `S-4`. Ce sont `S-4-P`, `S-4-1-P` et `S-4-1-2-P`.

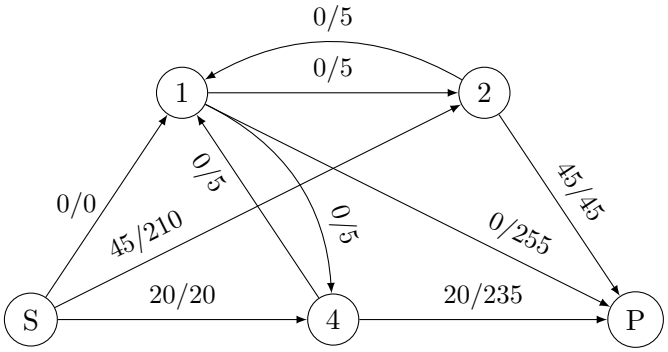


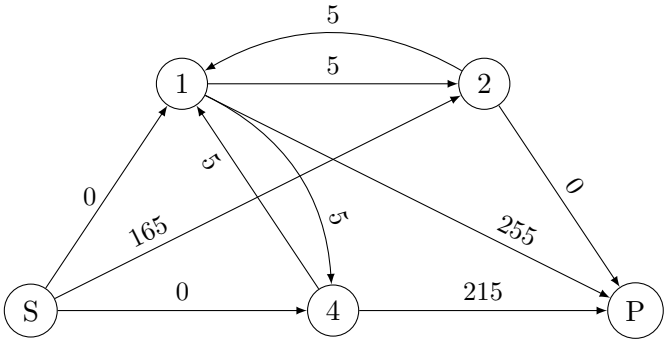
Les deux chemins les plus courts menant de `S` à `P` sont `S-2-P` (augmentation du flot de 45) et `S-4-P` (augmentation de 20). On choisit celui permettant une augmentation maximale, soit `S-2-P`, pour une augmentation du flot de 45.



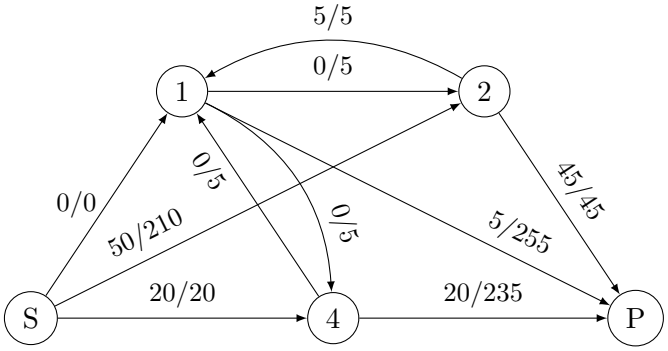


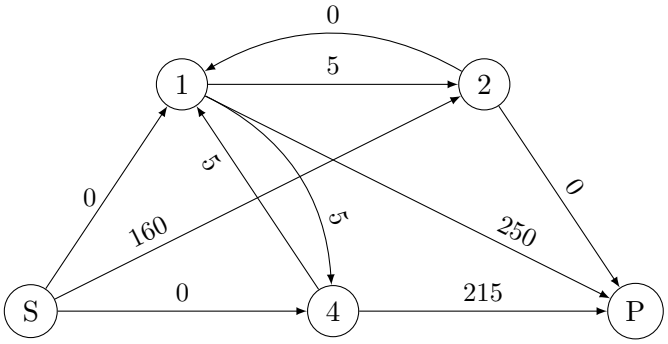
On prend alors l'autre chemin de longueur 2, `S-4-P`, qui augmente le flot de 20.





On a épuisé les chemins de longueur 2. Parmi les chemins de longueur 3, le seul à n'être pas saturé est `S-2-1-P`, que l'on choisit donc et qui augmente le flot de 5.





On constate que tous les chemins de `S` à `P` sont saturés. On est donc arrivé au bout de l'exécution de l'algorithme, pour un flot de $45 + 20 + 5 = 70$.

Q 14. Par « sommets accessibles », il faut comprendre *sommets autres que le puits accessibles par un chemin non saturé, c'est-à-dire dont aucune arête n'est saturée.* On s'appuie sur le dernier graphe obtenu à la question précédente. De la source `S`, on ne peut aller que vers le sommet `2`, les arêtes menant à `1` et `4` étant saturées. Du sommet `2`, on ne peut pas aller vers `1` car l'arête est saturée. On a donc $A = \{\texttt{S}, 2\}$, d'où $B = \{1, 4, \texttt{P}\}$. La capacité correspondante est

$$c(2,1) + c(2,P) + c(S,4) + c(S,1) = 5 + 45 + 20 + 0 = 70.$$

On retrouve bien le résultat obtenu à la question précédente.

Q 15. Conformément à l'exemple développé à partir de la Figure 10, les pixels noirs de départ sont non accessibles. La partie A correspond aux pixels devenus blancs et la partie B à ceux devenus noirs. On constate un résultat de bonne qualité : la lettre t est facilement reconnaissable.

Partie III. — Détermination des caractères

III.1. Analyse de la base données des caractères

Pour les requêtes, on passe le nom des tables en minuscules pour bien faire la différence avec les commandes SQL, écrites de manière standard en minuscules. C'est une question de lisibilité, pas de syntaxe, SQL n'étant pas sensible à la casse.

Q 16. `SELECT id FROM Fontes WHERE (nom = "Zurich" AND style = "romain" AND 10 <= taille AND taille <= 16)`.
Contrairement à python, on ne peut pas écrire `10 <= taille <= 16`.

Q 17.
```
SELECT c.fichier FROM Caracteres AS c JOIN Symboles as s ON c.id_symbole = s.id
WHERE s.label = "A".
```

Q 18.
```
SELECT label, COUNT(*) FROM
(Caracteres as c JOIN fontes as f ON c.id_fonte = f_id)
JOIN Symboles as s ON c.id_symbole = s_id
WHERE f.nom = "Zurich" AND f.style = "roman" AND f.taille >= 10 AND f.taille <= 16
GROUP BY s.label
```

III.2. Classification automatique des caractères

Q 19. La méthode `split` scinde la chaîne de caractères, selon la sous-chaîne en argument (ici, le caractère _) et crée une liste des sous-chaînes obtenues. Ainsi, `car = ['Zurich Light BT', 'majuscules18', '10.png']`.

Comme `car[2]` contient la chaîne `'10.png'`, on a `car[2].split('.') = ['10', 'png']`, d'où `num = '10'`.

Par slicing, on a `var = 'majuscules'` et la récupération de l'indice donne `ind = 0` car la première entrée de la liste `categories` est `'majuscules'`.

Enfin, la fonction renvoie la dixième majuscule, donc le caractère `'J'`.

Q 20. On crée le dictionnaire vide. À chaque nouveau nom de fichier, on crée l'entrée du dictionnaire si le caractère correspondant n'y figure pas déjà et on l'actualise dans le cas contraire.

```
def lire_donnees_ref(fichiers_car_ref: list) -> dict:
    carac_ref = {}
    for nomFichier in fichiers_car_ref:
        symbole = lire_symbole_fichier(nomFichier)
        if symbole in carac_ref:
            carac_ref[symbole].append(imread(nomFichier))
        else:
            carac_ref[symbole] = [imread(nomFichier)]
```

Q 21. Si l'on utilise `numpy`, on peut utiliser `np.sqrt` plutôt que de charger `sqrt` à partir de la bibliothèque `math`. Notons en passant que les valeurs de $p_{i,j}$ et $q_{i,j}$ valent 0 ou 1 et que le carré ne sert pas à grand chose. La norme 1 donne la même valeur. Là aussi, si l'on utilise `numpy`, on peut aller plus vite en utilisant les fonctions de cette bibliothèque, mais ce n'est pas ce qui est attendu ici, l'usage de `numpy` étant autorisé partout... sauf en informatique.

```
from math import sqrt

def distance(im1: array, im2: array):
```

```
    n0, n1, _ = dimension(img)
    d = 0
    for i in range(n0):
        for j in range(n1):
            d += (im1[i][j] - im2[i][j])**2
    return sqrt(d)
```

Q 22. On peut utiliser les compréhensions :

```
def calcul_distances(carac_ref: dict, carac_test: array) -> dict:
    return {symbole: [distance(carac_test, img) for img in carac_ref[symbole]] for symbole in carac_ref}
```

Ou faire des boucles ; c'est la même chose en plus aéré :

```
def calcul_distances2(carac_ref, carac_test):
    distances = {}
    for symbole in carac_ref:
        distances[symbole] = []
        for img in carac_ref[symbole]:
            distances[symbole].append(distance(img, carac_test))
    return distances
```

Q 23. La méthode au programme la plus performante dans le pire des cas est le *tri fusion* (*merge sort*). Rappelons que, dans le pire des cas, le tri rapide est quadratique (il n'est semi-linéaire qu'en moyenne).

Q 24. La fonction initialise la liste à renvoyer ; `voisins`, par une liste de couples contenant une valeur et un symbole, appelé lettre dans le code, la valeur étant la distance de l'image à l'une des représentations de la lettre. Les distances sont classiquement initialisées à l'infini, ce qui permet de faire rentrer dans la liste au départ n'importe quel élément. On parcourt ensuite les entrées du dictionnaire `distances` et, pour chacune de ces entrées, la liste des distances qu'elle contient. À chaque instant, la liste `voisins` contient les K plus proches voisins parmi ceux qui ont été déjà testés, liste complétée tant qu'on n'a pas encore testé K valeurs par des valeurs infinies. La plus grande valeur contenue dans la liste est ainsi celle d'indice $K-1$. Si l'élément lu correspond à une distance inférieure, on l'insère dans la liste sans échanger les éléments comme on le fait classiquement pour trier une liste en place, mais en décalant le dernier élément lu, ce qui permet de remplacer les valeurs devenues trop grandes.

```
def Kvoisins(distances: dict, K : int) -> list:
    voisins = [(float("inf")," ") for k in range(K)]
    for lettre in distances:
        d = distances[lettre]
        for j in range(len(d)):
            if voisins[K-1][0] > d[j]:
                k = len(voisins) - 1 # ou k = K - 1
                while k > 0 and voisins[k-1][0] > d[j] :
                    voisins[k] = voisins[k-1]
                    k = k-1
                voisins[k] = [d[j], lettre]
    return voisins
```

Q 25. Dans le pire des cas, on parcourt à chaque fois toute la liste des voisins. Les opérations effectuées étant de complexité constante, la complexité est en $\mathcal{O}(Kn)$. Par rapport au tri fusion, c'est intéressant si K est petit devant $\ln n$, ce que l'on peut considérer comme vrai (c'est ce qu'attend probablement l'énoncé), bien que ce soit discutable vu la très lente croissance du logarithme. Il faudrait aller voir du côté des constantes du $\mathcal{O}$ pour en être sûr ou, surtout, faire des tests.

Q 26. On crée un dictionnaire des lettres présentes dans les voisins et on en dénombre les occurrences.

```
def symbole_majoritaire(voisins: list) -> str:
    Lettres = {}
    for entree in voisins:
```

```
        lettre = entree[1]
        if lettre not in Lettres:
            Lettres[lettre] = 1
        else:
            Lettres[lettre] += 1
    record = 0
    for lettre in Lettres:
        if Lettres[lettre] > record:
            record, reference = Lettres[lettre], lettre
    return reference
```

On obtient un code un peu plus court en parcourant deux fois la liste voisins pour initialiser le dictionnaire avec ses clefs et des valeurs nulles. C'est indifférent vu que K est petit.

Q 27. Le nombre d'images de référence semble avoir un impact déterminant sur la qualité de la reconnaissance, ce qui n'est pas le cas du choix du nombre K de voisins.