I. Acquisition d'un document

1 Chaque bit peut prendre, indépendamment des autres, l'une des deux valeurs 0 ou 1. On peut donc donner à N bits distincts 2^N états possibles. La représentation binaire des nombres entiers permet d'associer chacun de ces états à un nombre entier,

> Il suffit donc de 8 bits pour représenter $2^8 = 256$ valeurs, soit par exemple chaque entier entre 0 et 255 inclus.

Les pixels sont des carrés dont la mesure du côté est définie par la résolution en ppp. Le nombre de pixels dans la hauteur de l'image peut être calculé par

$$p = \frac{29{,}7 \text{ cm} \cdot 300 \text{ pixels/pouce}}{2{,}5 \text{ cm/pouce}} = 3\,564 \text{ pixels}$$

De même, dans la largeur on a

$$q = \frac{21 \text{ cm} \cdot 300 \text{ pixels/pouce}}{2{,}5 \text{ cm/pouce}} = 2\,520 \text{ pixels}$$

Par conséquent, le nombre total de pixels d'une image au format A4 est le produit de p et q, et chacun nécessite un triplet de mots de 8 bits. La taille totale de l'image en bits est donc

$$3 \times 8 \times p \times q = 3 \times 8 \times 3\,564 \times 2\,520 = 215\,550\,720 \text{ bits}$$

2 Analysons la fonction pour déterminer sa complexité : la fonction `dimension` appelée à la première ligne est sûrement de complexité constante, alors que la fonction `initialise` s'exécute au pire en un nombre d'opérations proportionnel au produit $p \cdot q$ puisqu'il s'agit d'initialiser tous les pixels. Dans les lignes suivantes, on observe deux boucles imbriquées, l'extérieure bouclant sur `n0` soit p valeurs, et l'intérieure sur `n1` soit q valeurs. Le corps de la boucle comportant un nombre constant d'opérations élémentaires, la complexité totale des boucles est donc en $O(p \cdot q)$. Il en résulte que

> La complexité totale de la fonction `conversion_gris` est elle aussi en $O(p \cdot q)$.

> Notons la présence d'une coquille dans le code donné par l'énoncé, à la troisième ligne, qui aurait dû utiliser les variables `n0` et `n1` au lieu de p et q, non définies dans le programme.

3 Le tableau retourné par la fonction `conversion_gris` contient un entier par pixel, qui peut prendre des valeurs de 0 à 255 incluses. Une implémentation possible de la fonction `binarisation` est de boucler sur chaque ligne et colonne, et d'assigner au pixel correspondant la valeur 255 si sa valeur initiale est supérieure au seuil, et 0 sinon. Le reste est similaire à la fonction `conversion_gris`.

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

II. Reconnaissance du document

4 Procédons par élimination pour sélectionner la fonction correcte. La fonction en bas à droite ne fait pas d'interpolation selon `x`, on peut donc l'éliminer d'office. La fonction en bas à gauche, en plus d'autres problèmes, fait sa dernière interpolation entre `x0` et `y1`, ce qui n'a pas de sens. Il reste les deux fonctions du haut. Les deux premières interpolations de celle d'en haut à gauche sont faites en diagonale, puisque les indices à la fois de `x` et de `y` changent. C'est incorrect : l'interpolation linéaire consiste à d'abord interpoler dans une direction, puis dans l'autre.

La solution correcte est donc l'implémentation en haut à droite.

5 Considérons d'abord l'initialisation des variables `imr`, `angr` et `matR`. La fonction `rotation` renvoyant la variable `imr`, il s'agit de l'image après rotation, que l'on doit initialiser comme une image de dimensions `(p, q, 1)` où chaque pixel a la valeur 255. La variable `angr` indique par son nom qu'il doit s'agir de la valeur de l'angle transformée en radians. Quant à la variable `matR`, on la comprend comme une matrice de rotation que l'on peut définir en suivant l'énoncé.

> L'énoncé définit la matrice de rotation à l'envers : la matrice de rotation d'un angle α s'exprime en réalité
>
> $$\begin{pmatrix}\cos(\alpha) & -\sin(\alpha)\\ \sin(\alpha) & \cos(\alpha)\end{pmatrix}$$
>
> Celle que l'énoncé définit effectue donc une rotation d'un angle $-\alpha$. Or, par coïncidence, c'est ce qu'on veut ici, puisque pour faire tourner l'image d'un angle α, on va chercher les valeurs des pixels qui étaient situés à un angle $-\alpha$. On peut donc au choix utiliser la matrice de l'énoncé avec l'angle `angr`, ou la formule correcte avec l'angle `-angr` : ces deux solutions sont équivalentes.

L'énoncé indique que cette approche requiert d'itérer sur chaque pixel de l'image transformée, on utilise donc les intervalles `range(p)` et `range(q)` pour les indices `ni` et `nj`, qui couvrent tous les pixels, et on anticipe qu'il faudra au sein des boucles donner une valeur au pixel `imr[ni][nj]`. Dans les boucles, on commence par calculer la position du pixel tourné d'un angle $-\alpha$ autour du centre de l'image en appliquant la matrice de rotation au vecteur `[ni - p//2, nj - q//2]`, puis on déplace ce vecteur tourné pour obtenir une position en pixels en référence au coin de l'image.

On vérifie enfin que ce pixel antécédent est bien dans les limites de l'image. Pour pouvoir définir `x1` et `y1` dans la fonction `bilineaire`, on doit avoir $0 \leqslant x < p-1$ et $0 \leqslant y < q-1$, ce que l'on peut définir tel quel en python. Finalement, on affecte au pixel `imr[ni][nj]` sa valeur calculée par la fonction `bilineaire`. Il est inutile d'écrire un cas `else`, puisqu'il s'agirait d'attribuer au pixel la valeur 255, qui la possède déjà du fait de l'initialisation de l'image.

> En l'absence d'indication de l'énoncé en ce qui concerne numpy, on peut considérer cette bibliothèque chargée de la manière standard, c'est-à-dire au moyen de l'instruction `import numpy as np`.

```
def rotation(im: array, angle: float) -> array:
    p, q, _ = dimension(im)
    imr = initialise(p, q, 255)
    angr = angle * np.pi / 180.0
```

```
    matR = np.array(
        [[np.cos(angr), np.sin(angr)], [-np.sin(angr), np.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 = x + p//2
            y = y + q//2
            if 0 <= x < p - 1 and 0 <= y < q - 1:
                imr[ni][nj] = bilineaire(im, x, y)
    return imr
```

6 Si l'on se limite à des entiers entre 0 et 255 pour représenter les valeurs des pixels, on a déterminé en question 1 qu'il suffisait de 8 bits pour représenter chaque pixel. Huit bits correspondent à un octet, unité de mémoire directement adressable par le processeur, ce qui facilite la gestion de la mémoire. En utiliser davantage serait inutile et conduirait à une image qui prendrait plus de place en mémoire que nécessaire, ce qui peut également conduire à des temps d'exécution plus importants.

> Le processeur opérant sur des mots d'un, deux, quatre ou huit octets, utiliser des unités de mémoire de longueurs différentes peut avoir un intérêt dans certaines situations très particulières, mais est sous-optimal en général puisque cela peut nécessiter des réalignements ou conversions.
>
> Par ailleurs, la contribution de la quantité de mémoire accédée sur le temps d'exécution d'un programme peut être non-linéaire, en conséquence du comportement de certains systèmes d'exploitation qui ne lisent pas la mémoire octet à octet mais page par page. Charger une page (une zone) de mémoire est relativement long, mais y accéder par la suite est rapide. Si l'on représente les pixels de l'image avec un encodage inutilement encombrant, l'image, potentiellement contenue dans une seule page, pourrait éventuellement en dépasser et nécessiter le chargement de plusieurs pages de mémoire au lieu d'une seule, ce qui peut fortement affecter le temps d'exécution.

Le type `uint8` ne représentant que des entiers positifs, il est impossible d'y stocker le résultat de la soustraction $18-23$. En fonction du processeur et de sa configuration, une telle opération peut provoquer une erreur, un avertissement, ou bien s'effectuer silencieusement en saturant (retournant 0) ou en effectuant l'opération modulo 256, ce qui permet de retourner $256-(18-23)$ qui est bien positif. Dans tous ces cas, la fonction `lineaire` telle qu'implémentée dans l'énoncé retourne une valeur fausse lorsque `pix1` $<$ `pix0`. Pour régler ce problème, il suffirait ici de convertir `pix0` et `pix1` en entiers signés, ou mieux, directement en nombres flottants puisque c'est le type de retour de la fonction, avant d'effectuer leur soustraction.

7 Une implémentation possible de la fonction `histo_lignes` peut commencer par créer une liste vide. Puis, pour chaque ligne, on peut parcourir toutes les colonnes et incrémenter un compteur à chaque pixel noir. Le compteur est ajouté en fin de liste à la fin du décompte de chaque ligne.

```
def histo_lignes(im: array) -> list:
    p, q, _ = dimension(im)
    histo = []
    for i in range(p):
```

```
        compteur = 0
        for j in range(q):
            if im[p][q] == 0:
                compteur += 1
        histo.append(compteur)
    return histo
```

8 L'énoncé demande de manière pas extrêmement claire que la première et la dernière ligne d'un bloc soient blanches. On peut confirmer cela en observant la liste d'exemple. Elle contient les deux paires consécutives `[73, 102]` et `[102, 132]`, la même chose se reproduisant pour les paires `[293, 322]` et `[322, 351]`. La seule manière d'avoir une ligne de fin d'un bloc répétée en ligne de début du bloc suivant est que chaque bloc commence et se termine par une ligne blanche. Il est à noter que c'est impossible à assurer dans le cas général, par exemple si la première ou la dernière ligne de l'image contient des pixels noirs, mais ignorons ce cas.

> Dans les cas un peu flous comme celui-ci, il est important de mentionner ce que l'on comprend de la question, et d'expliquer ce que l'on choisit d'implémenter, pour ne pas risquer de perdre des points en programmant quelque chose de légèrement différent de ce qui est attendu.

On interprète la variable `lignes` comme la liste des blocs car c'est ce que la fonction retourne, la variable `i` comme un indice qui parcourt les lignes de la liste en paramètre de la fonction, et la variable `deb` comme le début d'un bloc. On peut donc commencer par remplir la condition de la boucle `while`, qui doit vérifier que l'on ne dépasse pas le nombre de lignes, ainsi que l'incrément de `i` en fin de boucle.

La première condition peut se renseigner en s'aidant du commentaire. On cherche ici à détecter les débuts de blocs, puisqu'on voit également que l'on modifie la variable `deb`. Cela ne doit se produire que lorsque deux conditions sont réunies en même temps : on était sur une ligne blanche (`deb == -1`) et on arrive sur une ligne non-blanche (`liste[i] > 0`). Dans ce cas, on attribue à `deb` l'indice de la première ligne noire du bloc, à savoir `i`. On fera ensuite débuter le bloc par `deb-1`.

La deuxième condition cherche les fins de blocs, elle doit donc se déclencher lorsque l'on était sur une ligne non-blanche (`deb >= 0`), et que l'on arrive sur une ligne blanche (`liste[i] == 0`). On a alors trouvé un bloc complet et on peut donc l'ajouter à la liste des blocs, contenue dans la variable `lignes`. Il faut également réinitialiser la variable `deb` à la valeur `-1` pour indiquer que l'on peut chercher un nouveau bloc.

```
def detecter_lignes(liste: list) -> list:
    lignes = []
    i = 0
    deb = -1    # -1 tant qu'on parcourt des lignes de pixels blancs
    while i < len(liste):
        # début d'une suite de lignes contenant des pixels noirs
        if deb == -1 and liste[i] > 0:
            deb = i
        # fin d'une suite de lignes contenant des pixels noirs
        elif deb >= 0 and liste[i] == 0:
            lignes.append([deb - 1, i])
            deb = -1
        i += 1
    return lignes
```

Il est regrettable que ce programme ne fonctionne pas dans tous les cas : si la première/dernière ligne de l'image n'est pas blanche, le premier/dernier bloc ne commencera/terminera pas par une ligne blanche, voire sera ignoré en ce qui concerne le dernier bloc, en fonction de la manière dont on remplit les trous de la fonction. On peut également se demander quel est l'intérêt d'entourer les blocs par deux lignes blanches, au lieu de la convention de faire commencer un bloc sur la première ligne non-blanche et de le faire terminer par la première ligne blanche suivant le bloc. Cela permettrait par exemple d'itérer directement sur chaque ligne d'un bloc à l'aide de la fonction `range`, dont le premier argument est inclus et le second exclu.

Par ailleurs, l'usage d'une boucle `while` dans le cas où une boucle `for` pourrait faire l'affaire n'est pas recommandé. Enfin, on pourrait facilement rendre les noms de variables plus explicites : appeler `liste` une liste n'est que très peu informatif et déconseillé. En prenant en compte toutes ces améliorations, on aurait pu écrire la fonction suivante :

```
def detecter_lignes(lignes: list) -> list:
    blocs, deb = [], -1
    for i in range(len(lignes)):
        if deb == -1 and lignes[i] > 0:
            # début d'un bloc contenant des pixels noirs
            deb = i
        elif deb >= 0 and lignes[i] == 0:
            # fin d'un bloc contenant des pixels noirs
            blocs.append([deb, i])
            deb = -1
    if deb >= 0:
        # dernier bloc s'il n'est pas suivi d'une ligne blanche
        blocs.append([deb, len(lignes)])
    return blocs
```

9 En comparant l'intervalle de recherche avant et après une itération, on voit que dans tous les cas, que le maximum se situe en `ac`, en `c` ou en `cb`, sa taille est divisée par deux. L'algorithme terminant lorsque la taille de l'intervalle devient inférieure à un epsilon positif donné, et la série géométrique de facteur 1/2 tendant vers zéro, **L'algorithme termine donc toujours.** Il s'agit de la **méthode de dichotomie**, utilisée pour trouver l'optimum de fonctions scalaires convexes.

L'idée sous-tendant cette méthode, consistant à diviser en deux l'intervalle de recherche à chaque itération, est à la base de nombreux autres algorithmes. On peut citer la recherche d'un zéro d'une fonction en utilisant son signe avant et après, ou bien la recherche d'un élément dans une liste triée.

Le nombre d'itérations de l'algorithme est le nombre d'étapes nécessaires pour arriver à une taille inférieure à ε en divisant par deux à chaque fois un intervalle de taille initiale $b-a$. C'est donc l'entier n le plus petit tel que

$$\frac{b-a}{2^n} < \varepsilon \qquad \text{soit} \qquad 2^n > \frac{b-a}{\varepsilon}$$

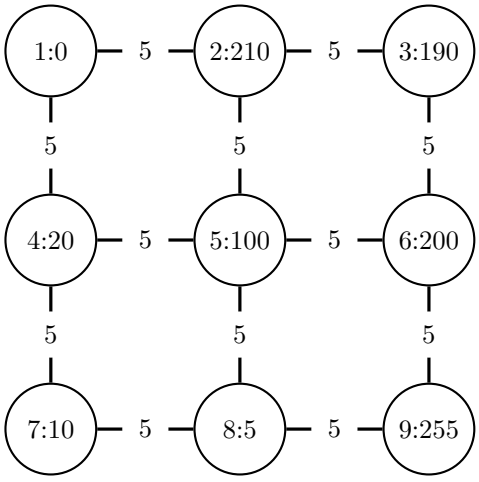
d'où

$$\boxed{n = \left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil}$$

10 L'essentiel de la fonction étant déjà rempli, il suffit de calculer les valeurs de `ac` et `cb` selon les indications de l'énoncé, puis de `fac` et `fcb`, qui, suivant le modèle de `fc`, représentent le nombre de zéros en ces points. Les conditions qui suivent servent à déterminer dans quel cas l'on se trouve, c'est-à-dire où est le maximum. Il suffit pour cela de regarder quelle valeur est attribuée à la variable `c`, soit le centre de l'intervalle de recherche : toujours la position du maximum. La première condition correspond donc au cas où le maximum est en `ac`, la seconde, qui ne modifie pas `c`, correspond au cas où `fc` est déjà le maximum, et la troisième au troisième cas. Il faut modifier dans ce cas les valeurs de `c` et `fc` en `cb` et `fcb`, ainsi que la variable `a` qui prend l'ancienne valeur de `c`.

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

11 L'énoncé précise que pour chaque sommet, il est créé une arête entre celui-ci et chacun de ses voisins. En lisant un peu plus loin, on comprend que le graphe décrit est non orienté. Par conséquent, on a le graphe suivant :

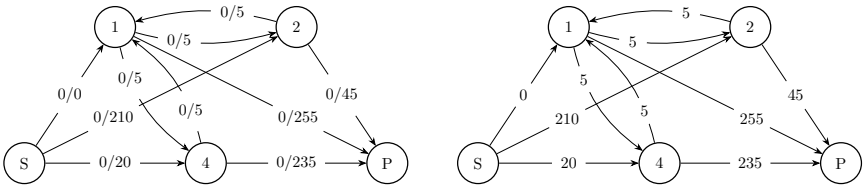


12 Les capacités entre la source et les nœuds sont les valeurs des pixels, celles entre les nœuds et le puits sont les compléments à 255 de ces valeurs, et les capacités entre nœuds sont 5 s'ils sont connectés et 0 sinon :

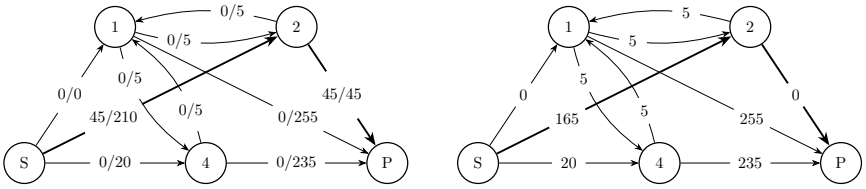
	S	1	2	3	4	5	6	7	8	9	P
S	0	0	210	190	20	100	200	10	5	255	0
1	–	0	5	0	5	0	0	0	0	0	255
2	–	–	0	5	0	5	0	0	0	0	45
3	–	–	–	0	0	0	5	0	0	0	65
4	–	–	–	–	0	5	0	5	0	0	235
5	–	–	–	–	–	0	5	0	5	0	155
6	–	–	–	–	–	–	0	0	0	5	55
7	–	–	–	–	–	–	–	0	5	0	245
8	–	–	–	–	–	–	–	–	0	5	250
9	–	–	–	–	–	–	–	–	–	0	0
P	–	–	–	–	–	–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

> On remarque qu'il s'agit presque de la matrice d'adjacence du graphe, qui est symétrique car le graphe est non orienté.

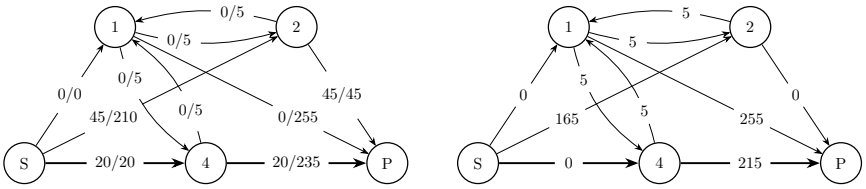
13 À chaque étape, il faut prendre le plus court chemin (en terme de nombre d'arêtes) qui a encore une capacité disponible non nulle.



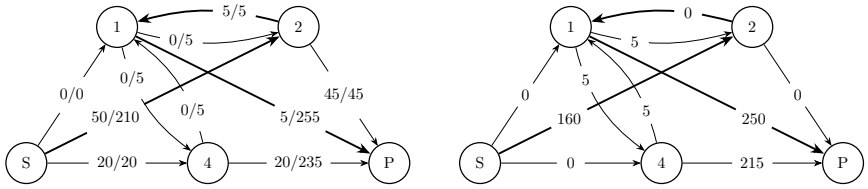
On choisit par exemple le chemin S-2-P, de longueur 2, et dont aucune des arêtes n'est nulle (on aurait pu aussi choisir S-4-P, qui est de même longueur, sans que cela ne change le résultat final de l'algorithme). La capacité minimale sur le chemin est de 45, c'est la valeur de l'augmentation du flot, qui sature l'arête 2-P.



On peut prendre ici l'autre chemin de longueur 2, S-4-P, qui a une capacité maximale de 20, et qui sature l'arête S-4.



Le chemin augmentant le plus court est ici S-2-1-P, de capacité maximale 5, et qui sature l'arête 2-1.



Il n'existe plus de chemin entre S et P qui ne contienne pas d'arête saturée, l'algorithme est donc terminé.

14 L'ensemble A, contenant la source et tous les sommets accessibles par celle-ci par des arêtes non saturées, contient dans le cas de la question précédente **la source S et le sommet 2**. L'ensemble B, complémentaire de l'ensemble A, contient **le puits P et les sommets 1 et 4**.

Le flot maximal est le plus facilement lisible en additionnant les capacités des arêtes sortant de la source (ou entrant dans le puits) : $0+50+20 = 45+5+20 = 70$.

La capacité de la coupe entre les ensembles $\mathrm{A} = \{\mathrm{S}, 2\}$ et $\mathrm{B} = \{1, 4, \mathrm{P}\}$ est la somme des capacités maximales des arêtes dirigées de A vers B. **Il s'agit donc de $(0 + 20) + (5 + 45) = 70$ également.**

15 Dans l'exemple avec les sommets 1, 2 et 4, on a obtenu un ensemble A qui contenait la source et le pixel 2, qui était le plus clair, et un ensemble B contenant le puits et les pixels 1 et 4, qui étaient les plus foncés. On peut donc deviner que

> L'ensemble B contient en général, en plus du puits, l'ensemble des pixels considérés comme appartenant au caractère, alors que l'ensemble A contient, en plus de la source, l'ensemble des pixels considérés comme appartenant à l'arrière-plan de la feuille.

Cet algorithme effectue une **segmentation binaire** de l'image. En l'absence d'arêtes entre les pixels voisins, ou si les capacités de ces arêtes étaient toutes nulles, l'algorithme attribuerait à A (arrière-plan) les pixels de valeur supérieure ou égale à 128, et à B (caractère) ceux de valeur inférieure ou égale à 127. Dans le cas contraire, si les capacités entre les pixels étaient infinies, on pourrait construire un flot entre la source et le puits qui sature complètement toutes les arêtes de l'un des deux (celui ayant la capacité totale minimale), tous les pixels appartiendraient donc à un seul ensemble.

On voit donc qu'en faisant varier les capacités entre pixels, on peut aller d'un extrême qui consiste à attribuer chaque pixel uniquement selon sa valeur, à un autre extrême qui ne considère que la connectivité des pixels.

Cet algorithme permet donc de segmenter l'image sur la base d'une combinaison entre valeurs des pixels et influence des voisins. En pratique, on peut considérer que les capacités entre voisins représentent une pénalité ajoutée au cas où deux pixels voisins seraient attribués à deux ensembles différents.

III. Détermination des caractères

16 Cette requête ne concerne que la table `FONTES` et consiste en une seule commande `SELECT`.

```
SELECT id FROM FONTES
WHERE nom = "Zurich"
  AND style = "romain"
  AND 10 <= taille
  AND taille <= 16
```

> On rappelle que la casse n'intervient pas en SQL, et on n'aurait pas faux si on écrivait les noms des tables en minuscules.

17 Cette requête nécessite d'associer des informations des tables `SYMBOLES` (pour les labels) et `CARACTERES` (pour les noms de fichiers). Il faut effectuer une jointure sur `SYMBOLES.id` et `CARACTERES.id_symbole`.

```
SELECT CARACTERES.fichier
FROM CARACTERES
  JOIN SYMBOLES ON CARACTERES.id_symbole = SYMBOLES.id
WHERE SYMBOLES.label = "A"
```

18 Ici, on doit associer des informations des trois tables. Deux jointures sont nécessaires, et puisqu'on veut un compte, on utilise la fonction d'agrégation `COUNT` associée à `GROUP BY`.

```
SELECT SYMBOLES.label, COUNT(CARACTERES.id)
FROM CARACTERES
  JOIN FONTES ON CARACTERES.id_fonte = FONTES.id
  JOIN SYMBOLES ON CARACTERES.id_symbole = SYMBOLES.id
WHERE FONTES.nom = "Zurich"
  AND FONTES.style = "romain"
  AND 10 <= FONTES.taille
  AND FONTES.taille <= 16
GROUP BY SYMBOLES.label
```

19 La variable `car` est une liste de chaînes de caractères. Ces chaînes sont celles qui étaient séparées par des caractères `_` dans le nom de fichier initialement contenu dans `nomFichier`. La variable `car` a donc trois éléments et représente donc ici la liste `["Zurich Light BT", "majuscules18", "10.png"]`.

La variable `num` est une chaîne de caractères, ceux qui sont situés avant le premier point dans le dernier élément de la liste `car`. Il s'agit donc de la chaîne `"10"`.

La variable `var` est une chaîne de caractères, qui correspond à ceux de l'élément d'indice 1 de la liste `car`, sauf les deux derniers. Ici, cela correspond à la chaîne `"majuscules"`.

Quant à la variable `ind`, elle représente le premier indice de la liste `categories` dont l'élément est égal à `var`, ici il s'agit de la position 0.

Enfin, la fonction retourne le caractère numéro `num` de l'élément numéro `ind` de la liste globale `symboles`, ici le caractère numéro 10 de la chaîne qui contient toutes les lettres majuscules, soit le caractère `"K"` (le premier caractère a un indice 0).

20 Après avoir créé le dictionnaire de retour, on itère sur la liste des fichiers, on utilise la fonction `lire_symbole_fichier` pour extraire la clé du dictionnaire et on lit le fichier à l'aide de la fonction `imread` pour obtenir sa valeur. Si le symbole n'a jamais été inséré dans le dictionnaire, on crée une liste vide. On peut ensuite ajouter le tableau à la liste de ceux associés à la lettre dans le dictionnaire.

```
def lire_donnees_ref(fichiers_car_ref: list) -> dict:
    carac_ref = {}
    for nomFichier in fichiers_car_ref:
        symbole = lire_symbole_fichier(nomFichier)
        image = imread(nomFichier)
        if symbole not in carac_ref:
            carac_ref[symbole] = []
        carac_ref[symbole].append(image)
    return carac_ref
```

21 Une implémentation élémentaire consiste à itérer sur chaque pixel de l'image, accumuler la somme de leurs différences au carré dans une variable dont on prend ensuite la racine carrée :

```
def distance(im1: array, im2: array) -> float:
    p, q, _ = dimension(im1)
    somme = 0
    for i in range(p):
        for j in range(q):
            somme += (im1[i][j] - im2[i][j]) ** 2
    return somme ** 0.5
```

> Une implémentation plus rapide, exploitant le fait qu'on stocke des tableaux numpy et que l'on peut donc utiliser des fonctions compilées, aurait été la suivante, dans laquelle on prend également soin de convertir les nombres en flottants avant de les soustraire, pour éviter les problèmes mentionnés à la question 6.
>
> ```
> def distance(im1: array, im2: array) -> float:
>     diff = im1.astype(np.float32) - im2.astype(np.float32)
>     return np.sqrt(np.sum(diff ** 2))
>     # ou encore np.linalg.norm(diff)
> ```

22 Dans cette fonction, on itère sur le dictionnaire `carac_ref` et on en crée un nouveau avec les mêmes clés et où chaque image est remplacée par la distance entre cette image et `carac_test`.

```
def calcul_distances(carac_ref: dict, carac_test: array) -> dict:
    carac_dist = {}
    for symbole in carac_ref:
        images = carac_ref[symbole]
        distances = []
        for image in images:
            dist = distance(image, carac_test)
            distances.append(dist)
        carac_dist[symbole] = distances
    return carac_dist
```

> On aurait pu également utiliser des compréhensions de liste et de dictionnaire, syntaxes spécifiques à Python qui permettent de créer et ajouter des éléments à une liste ou un dictionnaire en une seule ligne :
>
> ```
> def calcul_distances(carac_ref: dict, carac_test: array)->dict:
>     return {
>         symbole: [distance(img, carac_test) for img in images]
>         for symbole, images in carac_ref.items()
>     }
> ```

23 Le tri fusion, avec une complexité temporelle en $O(n\log(n))$ dans le pire des cas, est envisageable.

24 La fonction `Kvoisins` prend un dictionnaire de distances, un entier `K` et retourne une liste des `K` distances les plus faibles, dans l'ordre croissant, associées à leur lettre, toutes polices et fontes confondues. On voit qu'elle itère sur toutes les lettres, et que pour chacune, elle crée la variable `d` contenant la liste de toutes les distances du caractère considéré aux différentes représentations de cette lettre dans les différentes fontes. La boucle `for` itère sur ces fontes avec la variable `j`, comme il est confirmé par l'indexation `d[j]` quelques lignes plus bas. On observe que sous une condition à déterminer, la paire lettre et distance considérée est insérée dans la liste des `K` plus faibles distances. Cela n'a d'intérêt que lorsque la distance en question est inférieure à au moins l'une des distances considérées comme étant les plus petites à ce moment donné. En d'autres termes, on n'insère une nouvelle distance dans la liste des `K` plus petites que lorsqu'elle est inférieure à la plus grande de celles-ci. Enfin, la boucle `while` qui suit consiste à trouver l'endroit d'insertion, en décrémentant la variable `k` jusqu'à trouver un voisin dont la distance soit inférieure, ou alors arriver au début de la liste.

```
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 d[j] < voisins[-1][0]:
                k = len(voisins) - 1
                while k > 0 and d[j] < voisins[k][0]:
                    voisins[k] = voisins[k - 1]
                    k = k - 1
                voisins[k] = [d[j], lettre]
    return voisins
```

25 Les deux boucles `for`, ensemble, itèrent sur tous les n caractères disponibles. Dans le pire des cas, les distances sont ordonnées à l'envers, ce qui fait que pour chacune, la condition se déclenche et la boucle `while` effectue $K-1$, soit de l'ordre de K itérations. On a donc dans le pire cas une complexité en $O(n\,K)$.

Même si le O n'est censé être interprétable qu'asymptotiquement puisqu'on ne connaît pas la constante de proportionnalité, par comparaison à la solution naïve qui consistait à trier complètement les n caractères et à sélectionner les K plus proches, en $O(n\log(n))$, on peut espérer un gain lorsque $\log(n)$ est plus grand que K, soit pour $n > 150$, ce qui est le cas à partir de deux ou trois fontes de 79 caractères chacune.

> On aurait pu également répondre à la question en expliquant que si K a une borne supérieure comme ici, alors on peut l'omettre de la complexité algorithmique asymptotique, et on obtient donc asymptotiquement $\mathrm{O}(n)$ qui est linéaire et donc meilleure que $\mathrm{O}(n\log(n))$.

26 Cette fonction est le cas idéal d'utilisation d'un dictionnaire. En itérant sur les K voisins, on maintient un dictionnaire à jour avec comme clés les symboles et comme valeurs leur nombre d'occurrences jusqu'à présent. À la fin, on renvoie la clé dont la valeur est la plus importante.

```
def symbole_majoritaire(voisins: list) -> str:
    compteur = {}
    for voisin in voisins:
        if voisin not in compteur:
            compteur[voisin] = 0
        compteur[voisin] += 1

    majoritaire = voisins[0]
    for symbole in compteur:
        if compteur[symbole] > compteur[majoritaire]:
            majoritaire = symbole

    return majoritaire
```

27 On voit que la performance de l'algorithme augmente au fur et à mesure que l'on apporte des fontes dans la base de données. En revanche, il semble que le nombre de voisins considéré ne change rien (sur ce seul mot) à la performance de l'algorithme, on devrait donc pouvoir se contenter d'une faible valeur de K.