Informatique PCSI
TP 7 : algorithmes (manipulations d'une image)

Pour tester les programmes on utilise une image au format pbm.

Énoncé des exercices

Exercice 1

Écrire une fonction `inverser_nb` qui prend en paramètre une matrice représentant une image au format pbm et modifie la matrice afin d'obtenir une inversion noir et blanc.

```
def inverser_nb(img):
    haut, larg = len(img), len(img[0])
    ...
        ...
            ...
```

On dispose d'une image `chat.pbm`. On écrit les instructions :

```
mat = lire_fichier_pbm('chat.pbm')
inverser_nb(mat)
ecrire_fichier_pbm('inverse_chat.pbm', mat)
```

On obtient le résultat :

Exercice 2

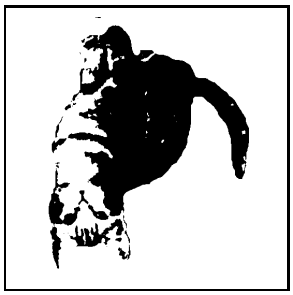
On envisage une symétrie axiale avec un axe vertical. Il convient donc d'échanger chaque pixel d'indice (i, j) avec le pixel d'indice $(i, p-1-j)$, i allant de 0 à $n-1$ et j allant de 0 à $p/2$. Attention à la boucle interne, si on écrit `for j in range(larg)` les pixels sont échangés deux fois et on obtient l'image initiale. L'indice i est l'indice de la ligne.

Écrire une fonction `sym_vert` qui prend en paramètre une matrice représentant une image au format pbm et modifie la matrice afin d'effectuer une symétrie d'axe vertical.

Pour une symétrie axiale avec un axe horizontal, il convient d'échanger chaque ligne d'indice i avec la ligne d'indice $n-1-i$ pour i allant de 0 à $n/2$. Ici aussi, attention à ne pas échanger deux fois les lignes avec une boucle `for i in range(haut)`.

Écrire une fonction `sym_hor` qui prend en paramètre une matrice représentant une image au format pbm et modifie la matrice afin d'effectuer une symétrie d'axe horizontal.

On obtient après une symétrie horizontale :



Exercice 3

Rotation

Pour une image représentée par une matrice de pixel, une rotation présente de nombreuses difficultés. Celles-ci proviennent principalement du fait qu'un pixel a des coordonnées entières. La suite se limite donc à des images carrées ($n = p$) avec une rotation d'un quart de tour.

Une rotation d'un quart de tour consiste à permuter les pixels des quatre quadrants. On note p_1 le pixel d'indice (i, j), p_2 le pixel d'indice $(n-1-j, i)$, p_3 le pixel d'indice $(n-1-i, n-1-j)$, p_4 le pixel d'indice $(j, n-1-i)$. On effectue pour i allant de 0 à $n/2$ et j allant de 0 à $n/2$, les permutations $(p_1, p_2, p_3, p_4) \longmapsto (p_2, p_3, p_4, p_1)$.

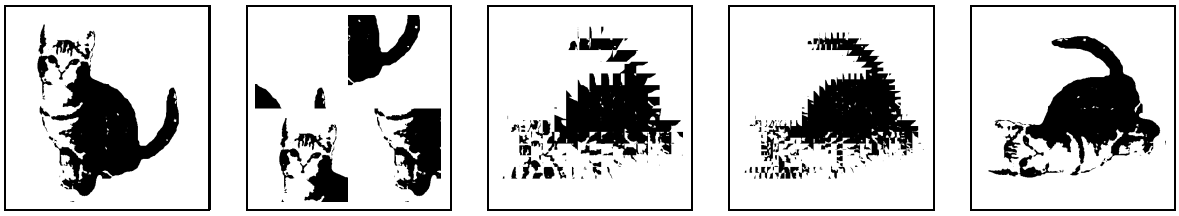
Divers algorithmes ont été étudiés dans les années 1980 pour manipuler des images en particulier pour un affichage correct sur un écran, à l'époque en noir et blanc. Avec peu de mémoire disponible et une capacité de calcul bien moindre qu'aujourd'hui, différentes *primitives* (des procédures de base) ont été développées comme « déplacer un bloc de bits d'un endroit à un autre en mémoire ». Un algorithme classique comme la rotation d'un quart de tour utilisait ce déplacement. Une image carrée est découpée en quatre carrés de même taille. Chaque carré est déplacé, $(c_1, c_2, c_3, c_4) \longmapsto (c_2, c_3, c_4, c_1)$ et on effectue la rotation d'un quart de tour de chaque carré par un appel récursif.

Compléter la fonction `rotation` qui suit :

```
def rotation(img, x, y, n): # n est la taille du carré
    if n > 1:
        n = n // 2  # pour découper le carré en 4 carrés
        # on déplace les carrés pixel par pixel
        for i in range(x, x+n):
            for j in range(y, y+n):
                temp = img[i][j]
                img[i][j] = img[...][...]
                img[i][n + j] = img[...][...]
                img[n + i][n + j] = img[...][...]
                img[n + i][j] = temp
        # on effectue les rotations des 4 carrés par appels récursifs
        rotation(img, x, y, n)
        rotation(img, ..., ..., ...)
        rotation(img, ..., ..., ...)
        rotation(img, ..., ..., ...)
```

On utilise la fonction `rotation` avec le fichier `chat.pbm`. La figure qui suit présente quelques résultats intermédiaires.

```
mat = lire_fichier_pbm('chat.pbm')
rotation(mat, 0, 0, len(mat))
ecrire_fichier_pbm('rot_chat.pbm', mat)
```



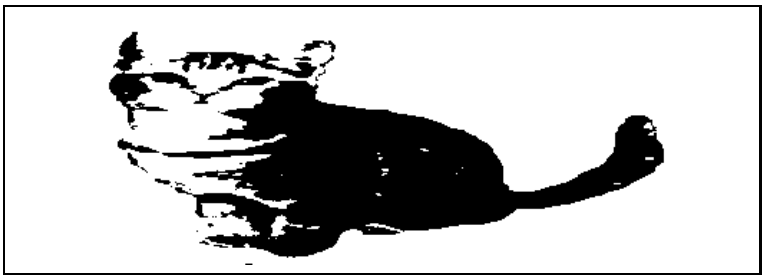
Exercice 4

Réduction et agrandissement

On dispose d'une image de taille (n, p) et on souhaite la réduire. Nous supposons que la réduction consiste à diviser la longueur et la largeur par un nombre entier d. C'est le cas le plus simple et on envisage de garder une ligne sur d lignes et une colonnes sur d colonnes.

```
def reduction(img, d):
    n, p = len(img), len(img[0])
    mat1 = [[0 for j in range(p)] for i in range(n//d + 1)]
    for i in range(n):
        if i % d == 0:
            for j in range(p):
                mat1[i//d][j] = img[i][j]
    mat2 = [[0 for j in range(p//d + 1)] for i in range(n//d + 1)]
    for i in range(n//d + 1):
        for j in range(p):
            if j % d == 0:
                mat2[i][j//d] = mat1[i][j]
    return mat2
```

La figure représente l'image intermédiaire avec $d = 3$.



Pour l'agrandissement par un facteur d entier, un procédé simple est de copier d fois chaque colonne puis d fois chaque ligne.

Écrire une fonction `agrandissement` en suivant le procédé décrit ci-dessus et le modèle de la fonction `reduction`.