Tle NSI Nom :…………………………………

Rotation d’un quart de tour

Dans ce problème, on considère une image « bitmap » implémentée par un tableau de tableau, les éléments du tableau étant des entiers indiquant le niveau de gris de l’élément. Chaque élément est appelé *pixel*[1] et on peut donc considérer l’image comme une matrice de pixels, chaque pixel étant un entier entre 0 et 255. Pour éviter les complications inutiles, on considère que l’image a autant de pixels en largeur qu’en hauteur[2] et de plus, on considère que cette dimension commune est une puissance de 2. Par exemple, voici une image 512×512 :

On aimerait faire tourner cette image d’un quart de tour dans le sens des aiguilles d’une montre.

1. Combien de pixels en tout, l’image comporte-t-elle ?

L’algorithme proposé fonctionne ainsi :

On commence par découper l’image 512×512 pixels en 4 images de 256×256 pixels, ainsi :

1 Pixel veut dire ***pic**ture **el**ement* soit image élémentaire (d’un seul niveau de gris)

2 On dit que la matrice est ***carrée***.

2. Combien de pixels chaque image 256×256 pixels comporte-t-elle ?

L’étape suivante consiste décaler les 4 images 256×256, chacune venant prendre la place de celle qui la suit dans le sens des aiguilles d’une montre :

- Le sud-ouest (l’œil gauche du chat) va au nord-ouest
- le nord-ouest (l’oreille gauche) va au nord-est
- le nord-est (l’oreille droite) va au sud-est
- le sud-est (l’œil droit) va au sud-ouest.

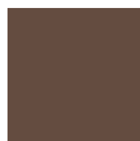
On obtient alors l’image suivante :

Ensuite, on ***fait faire un quart de tour*** à ***chacune*** des 4 images 256×256 :

Comme pour faire tourner l'image 512×512 pixel, on effectue 4 rotations des images 256×256 pixels, l'algorithme proposé est ***récursif***. Comme, de plus, les images 256×256 sont extraites de l'image 512×512, la méthode est ***diviser pour régner***. L'algorithme étant récursif, il faut déterminer le cas de base. Pour cela, on regarde un détail (au-dessus de l'œil gauche du chat) agrandi :

(l'image ci-dessus est de taille 32×32 pixels)

On voit que chaque pixel est uniformément colorié, par exemple le pixel en haut à gauche du détail ci-dessus ressemble à ceci :



On en déduit le cas de base : ***Tourner une image de 1×1 pixel, c'est ne rien faire*** du tout.

3. Pour évaluer la complexité de cet algorithme, on ne compte que les décalages. Par exemple, lors des premiers appels récursifs, on a décalé 4 images 256×256, ce qui représente 4

décalages. Pour tourner les images 256×256, on effectue 4 décalages d'images 128×128 pour chaque image 256×256, soit 4×4=16 décalages d'images 128×128 en tout. De même,

- On effectue en tout……… décalages d'images 64×64.
- On effectue en tout……… décalages d'images 32×32.
- On effectue en tout……… décalages d'images 16×16.
- On effectue en tout……… décalages d'images 8×8.
- On effectue en tout……… décalages d'images 4×4.
- On effectue en tout……… décalages d'images 2×2.

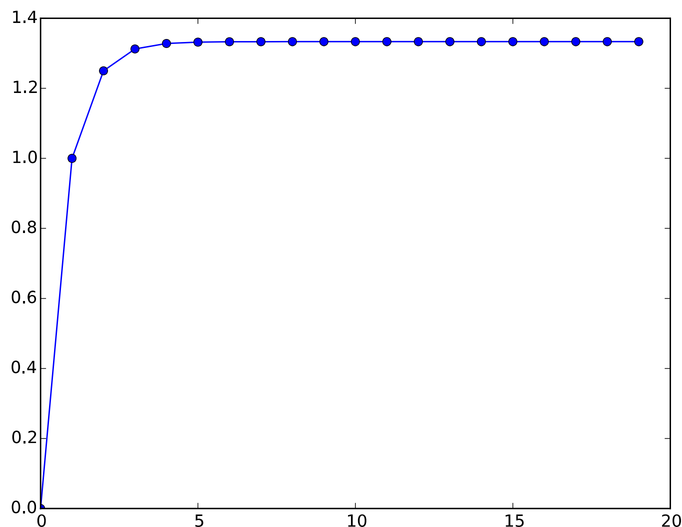
Combien de décalages effectue-on donc au total, pour tourner une image 512×512, avec cet algorithme récursif ?…………………………………………………………

On voudrait automatiser le décompte des déplacements (de pixels ou de blocs de pixels) nécessaires pour faire tourner d'un quart de tour, une image de dimensions $2^e \times 2^e$ où e est l'exposant. Pour cela on définit une fonction **ndep** (nombre de déplacements) dépendant de l'exposant **e**, ainsi :

```
def ndep(e) :
    if e==0 :
        return 0
    else :
         return 4*ndep(...........)+4
```

Compléter la fonction Python ci-dessus pour qu'elle renvoie le nombre de décalages nécessaires pour faire tourner une image de 2e pixels en largeur et en hauteur.

Pour faire tourner non récursivement une image de $2^e \times 2^e$ pixels, on a besoin d'effectuer $2^e \times 2^e$ déplacements (d'un pixel chacun). On rappelle que $2^e \times 2^e = 4^e$. Voici la représentation graphique de **ndep(e)/4**e** :



Peut-on considérer l’algorithme récursif comme efficace, par rapport à l’algorithme consistant à déplacer chaque pixel après avoir calculé ses nouvelles coordonnées ?

4. L’algorithme a été programmé ci-dessous en Python, sous la forme de la fonction **`rotation`**. Elle utilise le fait que l’import d’une image par la fonction **`imread`** de **`matplotlib.pyplot`** donne un tableau de tableaux.

```
from matplotlib.pyplot import *

def rotation(t) :
    dim = len(t)
    if dim == 1 :
        return t
    else :
        A = rotation([[t[i][j] for j in range(dim//2)] for i in range(dim//2)])
        B = rotation([[t[i][j] for j in range(dim//2, dim)] for i in range(dim//2)])
        C = rotation([[t[i][j] for j in range(..................)] for i in range(dim//2, dim)])
        D = rotation([[t[i][j] for j in range(..................)] for i in range(dim//2, dim)])
        t2 = []
        for i in range(dim//2) :
            t2.append([D[i][j] for j in range(dim//2)]+[A[i][j] for j in range(dim//2)])
        for i in range(dim//2) :
            t2.append([C[i][j] for j in range(dim//2)]+[B[i][j] for j in range(dim//2)])
        return t2


image = imread('lenna.png')
imshow(rotation(image), cmap=“gray”)
show()
```

Compléter la définition des variables **`C`** et **`D`** pour que la fonction **`rotation`** effectue bien un quart de tour.

Repère historique : Cet algorithme semble dû à Jeff Erickson, chercheur en géométrie algorithmique. Il l’a donné comme exercice à ses étudiants durant l’automne 2000.