PYTHON. **Compression des images. Corrigé**

1) Un pixel est caractérisé par une liste de 3 entiers. Pour ne pas choisir plusieurs fois le même pixel, il faut vérifier qu'il n'a pas déjà été choisi. Pour cela, on stocke les pixels choisis dans un dictionnaire dont la clé est le pixel, mais la clé ne peut être une liste.

On peut donc transformer $[r, g, b]$ en un triplet (r, g, b). *Remarque* : On pourrait aussi utiliser une chaîne de caractère (`str`), ou bien coder par l'entier $r + g \times 256 + b \times (256)^2$.

```
def nb_couleurs(img) :
    n = len(img) ; m = len(img[0])
    dico ={} # dictionnaire qui stocke les couleurs déjà rencontrées
    # clé du dictionnaire : les couleurs du pixel ; valeurs : nombre d'occurrence par exemple
    for i in range(n):
        for j in range(m) :
        L = img[i][j] ; c = (L[0],L[1),L[2])
        if c not in dico :    # accès en temps constant
            dico[c] = 1  # on ajoute la couleur au dictionnaire
        else :
            dico[c] += 1
    return len(d )
```

2)
```
def dist (p,q) :
    d = 0
    for i in range (3) :
        d = d +(q[i]-p[i])**2
    return d**0.5
```

3)

```
def initialise ( img , k ) :
    n = len(img) ; m = len(img[0])
    tab =[]
    dico = {} ; l = 0 # compteur
    while l <= k-1:
        i,j = random.randint(0,n),random.randint (0,n)
```

```
        L = img[i,j] ; c = (L[0],L[1),L[2])
        if c not in dico :  dico[c] = 1 ; l = l+1 ; tab.append(L)
    return (tab)
```

4)
```
def moyCouleur(L) :
    bary = [0]*3
    for v in L :
        for k in range(3) :
            bary[k] += v[k]
    for k in range(3) :
        bary[k] = bary[k]//len(L)
    return bary
def moyPixel(img,s) :
    L =  [ img[p[0],p[1]] for p in s
    return moyCouleur(L)
```

5)
```
def PlusProchePixel(p,mu) :
    dmin = float (" inf ")
    jmin =0
    for j in range(len(mu)) :
        d = dist(p,mu[j])
        if d < dmin :  dmin = d ; jmin = j
    return jmin
```

6)

```
def kmoyennes(img,k) :
    n = len(img) ; m = len(img[0])
    mu = initialise(img,k)
    test = True
    while test :
         part = [[] for i in range(k)]
         for i in range(n) :
              for j in range(m) :
```

```
                p = img[i][j]   # pixel correspondant
                ppp = PlusProchePixel(p,mu)
                part[ppp].append([i,j])
        ###
          new_mu = []
          for k_ind in range(k) :
          new_mu.append(moyPixel([img,part[k_ind]])]
        ###
          if new_mu == mu :  test = False
          mu = deepcopy(new_mu)
    return part
```

7)
```
def reduire(img,k) :
    img_new = deepcopy(img)
    part = kmoyennes(img,k)
    for k_ind in range(k):  # parcours des classes
         coul = moyPixel(img,part[k_ind])
         for p in part[k_ind] :  img_new[p[0]][p[1]] = coul
    return img_new
```

8)
```
def sectoriser(M,k) :
    if k ==0 :  return [M[0,0]]
    M0 = M[0:2**(k-1),0:2**(k-1)]
    M1 = M[0:2**(k-1),2**(k-1):2**k]
    M2 = M[2**(k-1):2**k,2**(k-1):2**k]
    M3 = M[2**(k-1):2**k,0:2**(k-1)]
    return [sectoriser(M0,k-1),sectoriser(M1,k-1),sectoriser(M2,k-1),sectoriser(M3,k-1)]
```

9)
```
def couleur(p,F,k) :
    (i,j) = p
    assert i<2**k and j<2**k
    if k ==0 :  return F[0]
    (a,b) = (i<2**(k-1),j<2**(k-1))
```

```
    if a and b :  return couleur((i,j),F[0],k-1)
    if a and not b :  return couleur((i,j-2**(k-1)),F[1],k-1)
    if not a and not b :  return couleur((i-2**(k-1),j-2**(k-1)),F[2],k-1)
    if not a and b :  return couleur((i--2**(k-1),j),F[3],k-1)
```

10)
```
def test(L):
    if len(L) != 4 :  return False
    for i in range(4) :
        if not isinstance(L[i],list) :  return False
        if not len(L[i]) == 1 :  return False
        if not isinstance(L[i][0],int) :  return False
    return True
```

11)
```
def ecart(L) :
    mu = 0 ; sigma = 0
    for i in range(4) :  mu = mu + L[i][0]
    mu = mu /4
    for i in range(4) :  sigma = sigma + (L[i][0]-mu)**2
    sigma = sigma ** 0.5
    return (sigma,round(mu))
```

12)
```
def comprime(F,delta)
    # On commence par un test d'arrêt.
    if len(F) == 1 :  return F
    # On utilise un parcours en profondeur se sorte à traiter d'abord les petits quadrants.
    M = [ comprime(G,delta) for G in F ]
    if test(M) :      # il est essentiel de placer ce test après les appels récursifs
        (sigma,r) = ecart(M)
        if sigma <= delta :  M = [r]
    return M
```