Corrigé : photomosaïque (Centrale 2020)

I Pixels et images

I.A – Pixels

Q 1. Un nombre entier positif codé sur 8 bits peut prendre $2^8 = 256$ valeurs distinctes, donc le nombre de couleurs que peut prendre un pixel est égal à $256^3 = 16\,777\,216$.

Q 2. Pour créer un pixel blanc on écrit `np.array([255, 255, 255], dtype=np.uint8)`

Q 3. On a $280 \equiv 24 \mod [266]$ donc $a = 24$. En revanche $240 \in [\![0, 255]\!]$ donc $b = 240$.
$24 + 240 = 264$ et $264 \equiv 8 \mod [256]$ donc $a + b = 8$.
On a $b \equiv -16 \mod [256]$ donc $a - b = 40$.
On a $0 \leqslant b < a$ donc $a // b = 0$.
Enfin, $a/b = 0,1$ (le résultat est un flottant).

Q 4. Il ne faut pas oublier de convertir le résultat en `np.uint8`.

```
def gris(p):
    return np.uint8(round(np.mean(p)))
```

I.B – Images

Q 5. `source` est un tableau de pixels à 3 000 lignes et 4 000 colonnes représentant une image en couleur. Le point situé en haut à gauche est un pixel de composantes RGB $(144, 191, 221)$.

Q 6.

```
def conversion(a):
    h, w, _ = a.shape
    g = np.zeros((h, w), dtype=np.uint8)
    for i in range(h):
        for j in range(w):
            g[i, j] = gris(a[i, j])
    return g
```

II Redimensionnement d'images

II.A – Le contexte

Q 7. Chaque vignette doit mesurer $\frac{2}{40} = 0,05$ mètres soit 50 millimètres de large. À raison de 10 pixels par millimètre cela donne $w = 500$, et donc $h = 375$ au ratio 4 : 3. La photomosaïque aura donc pour taille en pixels $W = 40w = 20\,000$ et $H = 40h = 15\,000$.

II.B – Algorithme d'interpolation au plus proche voisin

Q 8.

```
def procheVoisin(A, w, h):
    H, W = A.shape
    a = np.zeros((h, w), dtype=np.uint8)
    for i in range(h):
        for j in range(w):
            a[i, j] = A[(i * H) // h, (j * W) // w]
    return a
```

Q 9. Les opérations arithmétiques étant de coût constant, la complexité temporelle de cette fonction est en $O(wh)$.

II. C – Algorithme de réduction par moyenne locale

Q 10. La fonction `moyenneLocale` découpe l'image initiale en blocs de taille $pw \times ph$ avec $ph = H/h$ et $pw = W/w$ puis réalise la moyenne de chaque bloc pour obtenir le niveau de gris du pixel qui approchera ce bloc.

Q 11. Calculer la moyenne d'un bloc a un coût en $O(pw \cdot ph) = O\left(\frac{WH}{wh}\right)$; il y a wh blocs donc la complexité temporelle est en $O(WH)$.

II. D – Optimisation de la réduction par moyenne locale

Q 12. Chaque case de la table de sommation est une somme d'au plus N termes (le maximum étant obtenu pour $l = H$ et $c = W$) de type `np.uint8`. Le résultat est donc compris dans l'intervalle $[\![0, 255 \cdot N]\!]$.
Pour $N = 50\,000\,000$ on doit donc pouvoir représenter l'entier $12\,750\,000\,000$. Sachant que $2^{32} - 1 = 4\,294\,967\,295$, le type `np.uint32` n'est pas suffisant.

Q 13. Pour obtenir la complexité attendue, il faut bien évidemment ne pas recalculer les sommes à chaque fois, mais utiliser par exemple les formules :

$$\forall l \in [\![0, H-1]\!], \quad \forall c \in [\![0, W-1]\!], \qquad S(l+1, c+1) = S(l, c+1) + S(l+1, c) - S(l, c) + A(l, c)$$

```
1  def tableSommation(A):
2      H, W = A.shape
3      S = np.zeros((H + 1, W + 1), dtype=np.uint64)
4      for c in range(W):
5          for l in range(H):
6              S[l + 1, c + 1] = S[l, c + 1] + S[l + 1, c] - S[l, c] + A[l, c]
7      return S
```

Q 14. La ligne 8 calcule la somme des niveaux de gris des pixels compris dans le bloc `A[I:I+ph, J:J+pw]`; la ligne 9 affecte la moyenne de ces valeurs à l'image réduite.

Q 15. Ce calcul étant réalisé à coût constant, la complexité de cette fonction est proportionnelle au nombre de fois où elle est réalisée , soit un $O(hw)$.

Q 16. Le tableau `sred` découpe le tableau `S) en blocs de type \python{S[I: I+ph, J:J+pw]; dc` réalise les différentes opérations de type `S[I, J+pw] - S[I, J]`; enfin `dl` réalise les opérations de la ligne 8 de la fonction `reductionSommation1`, le tout de façon vectorielle.

Q 17. A priori, les deux versions ont même complexité temporelle, et une complexité spatiale moins bonne pour la seconde à cause des tableaux créés. Cependant, dans la pratique les opérations vectorielles sur les tableaux numpy sont très efficaces, et on peut donc espérer que la seconde version soit en réalité plus rapide que la première.

II. E – Synthèse

Q 18. `procheVoisin` présente l'avantage de pouvoir aussi agrandir une image, comme on le verra à la question 26. `reductionSommation` est avantageux si on veut calculer plusieurs réductions différentes de la même image, puisque la table de sommation n'est calculée qu'une fois.

III Sélection des images de la banque

III. A – Quelques requêtes

Q 19.

```
SELECT PH_id FROM Photo WHERE 3 * PH_larg = 4 * PH_haut
```

Q 20.

```
SELECT COUNT(*) FROM Photo JOIN Personne ON PH_auteur = PE_id
  WHERE PE_prenom = 'Alice' OR PE_prenom = 'Bernard'
```

Q 21.

```
SELECT PH_id, PH_date FROM Photo JOIN Decrit USING PH_id
                                 JOIN Motcle USING MC_id
  WHERE MC_texte = 'surf' AND EXTRACT(YEAR FROM PH_date) < 2006
```

Q 22.

```
SELECT PE_prenom, PH_id FROM Photo JOIN Present USING PH_id
                                   JOIN Personne USING PE_Id
  WHERE PH_auteur = PE_id
```

Q 23.

```
((SELECT PH_id FROM Photo JOIN Present USING PH_id
                          JOIN Personne USING PE_Id
    WHERE PE_prenom = 'Alice')
INTERSECT
(SELECT PH_id FROM Photo JOIN Present USING PH_id
                         JOIN Personne USING PE_Id
   WHERE PE_prenom = 'Bernard'))
EXCEPT
(SELECT PH_id FROM Photo JOIN Present USING PH_id
                         JOIN Personne USING PE_Id
   WHERE PE_prenom != 'Alice' AND PE_prenom != 'Bernard')
```

III. B – Internationalisation des mots-clés

Q 24. Je supprime la table `Motcle`, je fais de l'attribut `MC_id` la clé primaire de la table `Decrit`, et je crée la nouvelle table `Dialecte`, de clé primaire `D_id` :

Dialecte	
D_id	integer
MC_id	integer
D_langue	varchar(100)
D_texte	varchar(30)

Q 25.

```
SELECT PH_id FROM Photo JOIN Decrit USING PH_id
                        JOIN Dialecte USING MC_id
  WHERE D_langue = 'anglais' AND D_texte = 'mountain'
```

IV Placement des vignettes

IV. A – Préparatifs

Q 26. On peut utiliser la fonction `procheVoisin` :

```
def initMosaique(source, w, h, p):
    return procheVoisin(source, p * w, p * h)
```

Q 27. On utilise le type `int` pour la variable `s` pour éviter tout risque de débordement :

```
def L1(a, b):
    h, w = a.shape
    s = 0
    for i in range(h):
        for j in range(w):
            s += abs(a[i, j] - b[i, j])
    return s
```

Q 28.

```
def choixVignette(pave, vignettes):
    c, mc = 0, L1(pave, vignettes[0])
    for i in range(1, len(vignettes)):
        mi = L1(pave, vignettes[i])
        if mi < mc:
            c, mc = i, mi
    return c
```

IV. B – Méthode sans restriction du choix des vignettes

Q 29. Chaque pavé doit être remplacé par la vignette la plus proche.

```
def construireMosaique(source, vignettes, p):
    h, w = vignettes[0].shape
    mosaique = initMosaique(source, w, h, p)
    for i in range(p):
        for j in range(p):
            pave = mosaique[i * h: (i + 1) * h, j * w: (j + 1) * w]
            k = choixVignette(pave, vignettes)
            mosaique[i * h: (i + 1) * h, j * w: (j + 1) * w] = vignettes[k]
    return mosaique
```

Q 30. Le coût de `initmosaique` est un $O(wh)$ (question 9), le coût de `L1` est aussi en $O(wh)$ donc la complexité de `choixVignette` est un $O(whq)$. Le calcul du pavé et sa modification sont en $O(wh)$ donc la complexité totale est en $O\Big(wh + p^2(wh + whq)\Big) = O(p^2whq) = O(rnq)$ avec $r = p^2$ (le nombre de vignettes) et $n = wh$ (la taille des vignettes).

IV. C – Améliorations

Q 31. Si la banque d'image est suffisamment grande, on peut, au lieu de chercher la vignette la plus proche dans l'entièreté de la banque, ne la chercher que dans une fraction de celle-ci, cette part changeant à chaque fois. Dans la version de la question suivante, je ne prend en considération qu'un quart de la banque à chaque fois.

Q 32.

```
def construireMosaique(source, vignettes, p):
    h, w = vignettes[0].shape
    mosaique = initMosaique(source, w, h, p)
    for i in range(p):
        for j in range(p):
            pave = mosaique[i * h: (i + 1) * h, j * w: (j + 1) * w]
            k = choixVignette(pave, random.sample(vignettes, len(vignettes) // 4))
            mosaique[i * h: (i + 1) * h, j * w: (j + 1) * w] = vignettes[k]
    return mosaique
```