CORRIGÉ DM INFO n°4 (CENTRALE IPT 2020)

I. Pixels et images

❑ **Q1.** Un pixel a trois composantes chacune codée sur 8 bits soit 2^8 valeurs possibles pour chaque composante,et au total $(2^8)^3 = 2^{24} = 16\,777\,216$ couleurs possibles.

❑ **Q2.**

```
blanc = np.array([255, 255, 255], dtype = np.uint8)
# ou plus simplement:  np.array([255, 255, 255], np.uint8)
```

❑ **Q3.** Ici les calculs sont effectués modulo 2^8.

```
import numpy as np

a = np.uint8(280)
b = np.uint8(240)

print("a, b =", a, b) # 280 modulo 256 = 24

print("a+b =", a + b) # 24+240 = 264 et modulo 256 cela donne 18

print("a-b =", a - b) #24 - 240 = -216 et modulo 256 cela donne 40

print("a//b, a/b = ", a//b, a/b)
a, b = 24 240
a+b = 8
a-b = 40
a//b, a/b =  0 0.1
```

❑ **Q4.**

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

❑ **Q5.** `source.shape` donne les dimensions du tableau `numpy` `source`.
Ce tableau possède 3000 lignes et 4000 colonnes ; cela correspond respectivement à la hauteur H et à la largeur W de l'image (cela n'était pas dit dans l'énoncé), qui est donc au format $\frac{4}{3}$.
Le "3" correspond au nombre de composantes de chaque pixel ; ici l'image est en mode RGB (on peut aussi avoir comme résultat "4" si l'image est en mode RGBA).

La deuxième information de l'énoncé signifie que le pixel de coordonnées (0,0) (en haut à gauche) a pour triplet de couleurs R = 144 (composante rouge), G = 191 (composante verte) et B = 221 (composante bleue).

❑ **Q6.** Il y a plusieurs réponses possibles selon le degré de connaissances que l'on a de `numpy`. Mais je rappelle que `numpy` n'est PAS au programme, donc seule la 1ère réponse ci-dessous était exigible :

```
def conversion1(a:np.ndarray) -> image:
    H, W, _ = a.shape
    tab = np.zeros( (H, W), dtype = np.uint8)
    for i in range(H):
        for j in range(W):
            tab[i,j] = gris( a[i,j] )
    return tab

def conversion2(a:np.ndarray) -> image:
    # on vectorialise la fonction écrite en Q4 pour pouvoir
    # l'appliquer directement à un tableau numpy
```

```
    gris_vect = np.vectorize( gris )
    return gris_vect(a)

def conversion3(a:np.ndarray) -> image:
    # on utilise la fonction a.mean(d) décrite en fin de sujet
    return np.array(a.mean(2).round(), np.uint8)
```

:

II. Redimensionnement d'images

❏ **Q7.** La photomosaïque fait 2m de large, soit 2000mm, ou 20000 pixels. Comme il y aura 40 vignettes sur la largeur, cela fait 500 pixels de large par vignette et donc 375 pixels de haut ($\frac{3}{4} \times 500$).

La photomosaïque fera donc au total $40 \times 40 \times 500 \times 375 = 300\,000\,000$ pixels.

❏ **Q8.**

```
def procheVoisin(A:image, w:int, h:int) -> image:
    H, W = A.shape # ici l'image est en gris, il n'y a que 2 dimensions
    tab = np.zeros( (h,w), dtype = np.uint8)
    for i in range(h):
        i0 = (i*H) // h
        for j in range(w):
            j0 = (j*W) // w
            tab[i,j] = A[i0, j0]
    return tab
```

❏ **Q9.** La complexité de `np.zeros` est en $O(wh)$. Chaque passage dans les boucles `for` réalise un nombre constant d'opérations élémentaires, et il y a $w \times h$ passages, d'où une complexité en $O(wh)$ au total.

❏ **Q10.** La fonction `moyenneLocale` commence par créer un tableau vide qui va contenir l'image redimensionnée.

Lors des passages dans les deux boucles `for`, chacun des pixels de la nouvelle image prend comme valeur la moyenne des valeurs de l'image initiale correspondant aux pixels situés dans un rectangle de dimensions `ph` et `pw`. Plus précisément, le pixel de coordonnées `I // ph` et `J // pw` dans la nouvelle image correspond au rectange dont le coin en haut à gauche a pour coordonnées `I` et `J` dans l'image initiale, et de dimensions `ph` et `pw`.

❏ **Q11.** Chaque appel à la fonction `np.mean` réalise $O(ph \times pw)$ opérations. On effectue ce calcul $H/ph \times W/pw$ fois, d'où une complexité totale en $O(H \times W)$.

❏ **Q12.** Pour une image de 50 millions de pixels, chacun pouvant prendre des valeurs entre 0 et 255, la somme peut être égale au maximum à $50 \times 10^6 \times 255 = 1275 \times 10^7$.

Or su 32 bits, le plus grand entier non signé que l'on peut représenter vaut $2^32 - 1 = 4\,294\,967\,295$ qui est presque 3 fois inférieur au nombre précédent, donc le type `np.uint32` ne convient pas.

❏ **Q13.** On calcule ici les sommes au fur et à mesure, en réutilisant les sommes précédentes, après avoir remarqué que :

$$\sum_{\substack{0\leqslant i<\ell\\0\leqslant j<c}} A(i,j) = \sum_{\substack{0\leqslant i<\ell-1\\0\leqslant j<c}} A(i,j) + \sum_{\substack{0\leqslant i<\ell\\0\leqslant j<c-1}} A(i,j) - \sum_{\substack{0\leqslant i<\ell-1\\0\leqslant j<c-1}} A(i,j) + A(l,c).$$

Cette remarque permet d'assurer une complexité totale en $O(H \times W) = O(N)$.

```
def tableSommation(A:image) -> np.ndarray:
    H, W = A.shape
    tab = np.zeros(  (H+1, W+1), dtype= np.uint64)
    for i in range(1, H+1):
        for j in range(1, W+1):
            tab[i, j] = tab[i-1, j] + tab[i, j-1] - tab[i-1, j-1] + A[i, j]
    return tab
```

❏ **Q14.** La fonction `reductionSommation1` utilise la table de sommation `S` pour calculer les sommes sur chaque rectangle de dimension $ph \times pw$ et dont le coin en haut à gauche a pour coordonnées `I,J` dans l'image `A`. La formule utilisée (ligne 8) est similaire à celle utilisée dans la question précédente.

Cette somme calculée pour chaque petit rectangle est ensuite moyennée (division par le nombre de cases et arrondi) puis mise à la bonne place dans le tableau `a` (ligne 9).

❏ **Q15.** La boucle sur `I` est effectuée $h = H/ph$ fois, celle sur `J` l'est $w = W/pw$ fois, et les lignes 8 et 9 s'exécutent en temps constant ; la complexité temporelle est donc de $O(h \times w)$.

❏ **Q16.** Dans `reductionSommation1`, ligne 8, on remarque que les seuls éléments qui nous intéressent dans `S` sont ceux dont les indices sont des multiples de `ph` pour les lignes et de `pw` pour les colonnes. La ligne 4 crée alors le tableau `sred` ne comportant que ces valeurs.

La fin du calcul ligne 8 dans `reductionSommation1` effectue la différence des deux éléments de `S` situés sur la même ligne d'indice `I` et d'indices de colonnes `J+pw` et `J` ; cela revient donc à soustraire deux éléments d'indices consécutifs dans le tableau `sred` : c'est ce qui est fait ligne 5 dans la fonction `reductionSommation2` (je pense que le nom de variable : `dc` signifie « différence des colonnes » et `dl` « différence des lignes »). Ainsi `dc[I]` est la ligne contenant les éléments `S[I, J+pw] - S[I, J]` pour `J in range(0, H, ph)`.

La ligne 6 de la fonction `reductionSommation2` effectue alors le calcul des `sred[I+ph, ...] -sred[I, ...]`, c'est-à-dire termine le calcul de la ligne 8 de la fonction `reductionSommation1`

Enfin la ligne 7 de la fonction `reductionSommation2` calcule les moyennes en divisant par la taille des rectangles et la ligne 8 renvoie le tableau après avoir arrondi les valeurs précédentes.

❏ **Q17.** La complexité asymptotique temporelle de la seconde version est exactement la même que pour la première version, puisque les calculs sur les tableaux `numpy` font exactement les mêmes opérations.

La complexité spatiale est plus grande, puisque la fonction utilise 4 tableaux supplémentaires (`sred, dc, dl d`) alors que la première version n'utilise que le tableau `a`.

Cependant, l'intérêt de `réductionSommation` est qu'elle utilise des primitives `numpy` pour agir sur tout le tableau d'un coup. Or `numpy` utilise des primitives écrites en langage C qui sont bien plus rapides qu'un programme écrit en Python ; donc la seconde fonction est plus efficace que la première, même si elles sont de même complexité temporelle.

❏ **Q18.** Les programmes `moyenneLocale` et `réductionSommation` ne sont pas utilisables si la dimension de la nouvelle image ne divise pas l'ancienne ; il faut alors utiliser `procheVoisin`.

`moyenneLocale` et `tableSommation`, qui est un préalable à `réductionSommation` ont la même complexité totale en $O(H \times W)$. Cependant, une fois le tableau de sommation calculé, chaque redimensionnement suivant est en $O(h \times w)$ en utilisant seulement `réductionSommation`. C'est donc la méthode à privilégier si on pense effectuer plusieurs redimensionnement d'une même image à plusieurs échelles différentes.

Enfin, l'image est de meilleure qualité avec `réductionSommation` et `moyenneLocale` qu'avec `procheVoisin`.

III. Sélection des images de la banque

❏ **Q19.**

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

❏ **Q20.** Il faut faire ici une jointure des deux tables `Photo` et `Personne`. On peut écrire :

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

ou plus simplement :

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

❏ **Q21.** Il faut faire ici une jointure des trois tables `Decrit`, `Motcle` et `Photo`. Il faut faire attention au fait que certains attributs ont les mêmes noms dans différentes tables. De plus, il ne faut pas oublier de récupérer l'année dans `PH_date` grâce à `EXTRACT`.

On peut écrire :

```
SELECT PH_id, PH_date
FROM Decrit
    JOIN Photo ON Decrit.PH_id = Photo.PH_id
    JOIN Motcle ON Decrit.MC_id = MotCle.MC_id
WHERE MC_texte = 'surf' AND EXTRACT(year FROM PH_date) < 2006
```

ou bien, en utilisant les indications à la fin de l'énoncé :

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

ou encore, de façon plus basique (mais plus simple je trouve) :

```
SELECT PH_id, PH_date
    FROM Photo AS Pho, Motcle AS Mot, Decrit AS Decr
    WHERE Pho.PH_id = Decr.PH_id AND Mot.MC_id = Decr.MC_id
                AND MC_texte = 'surf'  AND EXTRACT(year FROM PH_date) < 2006
```

❏ **Q22.** On pourrait faire ici une jointure des trois tables `Photo`, `Present` et `Personne`, mais la table `Personne` doit être reliée à la table `Photo` d'une part, et `Present` d'autre part. Elle va donc figurer deux fois dans la requête.

```
SELECT PE_prenom, PH_id
FROM Photo
    JOIN Present ON Present.PH_id = Photo.PH_id
    JOIN Personne AS Photographe ON Photographe.PE_id = Photo.PH_auteur
    JOIN Personne AS Personnage ON Personnage.PE_id = Present.PE_id
WHERE Photographe.PE_id = Personnage.PE_id
```

ou :

```
SELECT PE_prenom, PH_id
FROM Photo, Present , Personne AS Photographe, Personne AS Personnage
    WHERE Photo.PH_id = Present.PH_id AND Present.PE_id = Personage.PE_id
     AND Photo.PH_auteur = Photographe.PE_id AND Photographe.PE_id = Personnage.PE_id
```

Mais il est beaucoup plus simple d'utiliser `INTERSECT` :

```
( SELECT PE_prenom, PH_id
    FROM Photo, Personne
    WHERE Photo.PH_auteur = Personne.PE_id )
INTERSECT
( SELECT PE_prenom, PH_id
    FROM Present, Personne
    WHERE Present.PE_id = Personne.PE_id )
```

❏ **Q23.** Première solution :

```
SELECT PH_id
    FROM Photo AS Pho, Present AS Pre, Personne AS Per
    WHERE Pho.PH_id = Pre.PH_id AND Pre.PE_id = Per.PE_id
        AND PE_prenom = 'Alice' OR PE_prenom = 'Bernard'
    GROUP BY Pre.PH_id HAVING COUNT(*) = 2
```

ou bien, en utilisant `EXCEPT` :

```sql
( SELECT PH_id
      FROM Photo AS Pho, Present AS Pre
      WHERE Pho.PH_id = Pre.PH_id )
EXCEPT
( SELECT PH_id
      FROM Photo AS Pho, Present AS Pre, Personne AS Per
      WHERE Pre.PE_id = Per.PE_id AND PE_prenom <> 'Alice' AND PE_prenom <> 'Bernard' )
```

(Rem : l'opérateur « différent » peut se noter indifféremment <> ou != comme en Python).

❏ **Q24.** On peut modifier uniquement la table `Motcle`, en rajoutant un champ `langage ( varchar(3) )`, et en donnant pour nouvelle clé primaire le couple `MC_id, langage`.

Les langues seront décrites avec trois lettres (par exemple ENG pour l'anglais, ESP pour l'espagnol etc...).

Chaque mot clé dispose d'un identifiant unique (le même qu'avant), mais le texte associé est désormais dépendant de la langue choisie.

❏ **Q25.** On s'inspire de **Q 21**.

```sql
SELECT PH_id
FROM Photo AS Pho, Motcle AS Mot, Decrit AS Decr
    WHERE Pho.PH_id = Decr.PH_id AND Mot.MC_id = Decr.MC_id
        AND Mot.MC_texte ='mountain' AND Mot.langage = 'ENG'
```

IV. Placement des vignettes

❏ **Q26.**

```python
def initMosaique(source:image, w:int, h:int, p:int) -> image:
    return moyenneLocale(source, w*p, h*p)
```

❏ **Q27.** On ne devait pas ici utiliser la fonction `valeur absolue`; en effet, pour des variables de type `np.uint8`, si $x \in [\![0;255]\!]$, `abs(-x)` n'est pas égal à x mais à $256-x$...

```python
def L1(a:image, b:image) -> int:
    somme = 0
    h, w = a.shape
    for i in range(h):
        for j in range(w):
            if a[i, j] > b[i, j]:
                somme += a[i, j] - b[i, j]
            else:
                somme += b[i, j] - a[i, j]
    return somme
```

❏ **Q28.** Une fonction classique de recherche de l'indice du minimum.

```python
def choixVignette(pave:image, vignettes:[image]) -> int:
    imin = 0
    dmin = L1(pave, vignettes[0])
    for i in range(1, len(vignettes)):
        dist = L1(pave, vignettes[i])
        if dist < dmin
            dmin = dist
            imin = i
    return imin
```

❏ **Q29.**

```python
def construireMosaique(source:image, vignettes:[image], p:int) -> image:
    h, w = vignettes[0].shape
    tab = initMosaique(source, w, h, p)
    for i in range(p):
        for j in range(p):
            pave = tab[i*h:(i+1)*h, j*w:(j+1)*w]
            ind = choixVignette(pave, vignettes)
            tab[i*h:(i+1)*h, j*w:(j+1)*w] = vignettes[ind]
    return tab
```

❏ **Q30.** L'appel à la fonction `initMosaique`, qui se contente d'appeler `moyenneLocale`, se fait avec une complexité de l'ordre de la taille de l'image `source` (cf. **Q 11**) c'est-à-dire $p^2 \times n$ (p^2 vignettes de tailles n).

Ensuite, il y a $r = p^2$ passages dans les deux boucles `for`, et chacun réalise une recherche de la meilleure vignette, ce qui demande $O(n \times q)$ opérations (q appels à `L1`, chacun étant en $O(n)$). Soit au total une complexité en $O(nqr)$.

La complexité totale est donc en $O\left(n(p^2+qr)\right)$.

❏ **Q31.** BOF....