DS4 - INFORMATIQUE
PHOTOMOSAIQUES
CORRIGE

Q1. On suppose que chacune des trois composantes RGB d'un pixel est représentée par un nombre entier positif ou nul, codé sur 8 bits. Combien de couleurs différentes peut-on représenter avec un tel pixel ?

Avec trois composantes (R, G et B) de chacune huit bits, on peut réaliser 2^8 x 2^8 x 2^8 = 2^{24}, soit 16 777 216 combinaisons.

Q2. Donner une instruction permettant de créer un vecteur correspondant à un pixel blanc.

Une instruction permettant de créer un vecteur correspondant à un pixel blanc peut être la suivante :

```
pixelBlanc = np.ones((3,), dtype=np.uint8)*255
```

Q3. Écrire une fonction d'entête `def gris(p:pixel) -> np.uint8:`

Le niveau de gris d'un pixel est défini ici comme étant la meilleure approximation entière de la moyenne des trois composantes RGB du pixel, la fonction calculant le niveau de gris peut donc s'écrire comme suit :

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

Q4. Interpréter ces valeurs.

L'instruction `source.shape` renvoie la forme du tableau numpy associé à l'image `surfer.jpg`, soit un tableau de 3000 lignes, 4000 colonnes et comportant 3 informations pour chaque élément du tableau. On en déduit alors que l'image possède 3000 pixels de haut et 4000 pixels de large, chaque pixel contenant les informations de couleur (RGB : *red*, *green* et *blue*).

L'instruction `source[0,0]` renvoie les valeurs de l'élément situé à la position `(0,0)` du tableau (en haut à gauche, *c.f.* Figure 2). Le pixel en haut à gauche de l'image `surfer.jpg` possède une intensité de rouge à 144 (0 étant une absence de couleur), une intensité de vert à 191 et une intensité de bleu à 221. Chaque couleur est codée par un octet et ces trois intensités sont regroupées dans un tableau de type `np.array`.

Q5. Écrire une fonction d'entête `def conversion(a:np.ndarray) -> image:`

```
def conversion(a):
    H, W = a.shape
    image = np.zeros((H,W),np.uint8)
    for i in range(0,H):
        for j in range(0,W) :
             image[i,j]=gris(a[i,j])
             #ou image[i,j]=np.uint8(round(a[i,j].mean()))
    return image
```

Q6. Quelle taille de vignette (w × h, en pixels) faut-il choisir ? Quelle sera alors la taille en pixels de la photomosaïque ?

La photomosaïque fait 40 vignettes de hauteur et 40 vignettes de largeur (40 x 40 = 1600), on sait également qu'elle fait deux mètres de large, donc une vignette fait 2/40 = 0,05 m, soit *w* = 50 mm de large. La résolution étant de 10 pixels par millimètre, une vignette fait 500 pixels de large.

Le ratio $W/H = w/h = 4/3$, permet d'obtenir la hauteur en pixel d'une vignette, soit $h = 3/4 \, . \, w$, l'application numérique donne une hauteur de vignette de 375 pixels de hauteur.

h = 375 pixels et w = 500 pixels

La photomosaïque fait 40 vignettes de hauteur et 40 vignettes de largeur, on a donc les dimensions suivantes pour la photomosaïque :
H_photomosaïque = 40 . h et W_photomosaïque = 40 . w

L'application numérique donne les dimensions :

H_photomosaïque = 15000 pixels et W_photomosaïque = 20000 pixels

Q7.Écrire une fonction d'entête `def procheVoisin(A:image, w:int, h:int) -> image:`

```
def procheVoisin(A, w, h):
    H, W = A.shape
    a = np.zeros((h,w),np.uint8)
    for i in range(0,h):
         for j in range(0,w) :
              a[i,j]=(A[int(i*(H/h)),int(j*(W/w))])
    return a
```

Q8.Quelle est sa complexité temporelle asymptotique ?

La complexité temporelle asymptotique est le nombre d'opérations élémentaire (affectations, comparaisons, opération arithmétiques) effectuées par l'algorithme. On émet l'hypothèse que toutes les opérations élémentaires sont à égalité de coût. L'analyse asymptotique donne une mesure de l'efficacité d'un algorithme, ne dépendant pas de constantes spécifiques à une machine. Ce nombre s'exprime en fonction de la taille *n* des données.

Soit une complexité temporelle asymptotique de : 2+1+h.w.7 + 1 = 3 + 7n = **0(n) avec : n= h.w**

Q9.Expliquer en quelques lignes son principe de fonctionnement.

La fonction `moyenneLocale` prend comme argument une image initiale et les dimensions de l'image réduite renvoyée par la fonction. Elle segmente l'image initiale en fonction de la taille de l'image finale à l'aide de divisions entières et chaque pixel de l'image finale est la moyenne locale du segment de l'image initiale associé.

Q10. Donner sa complexité temporelle asymptotique.

Sa complexité temporelle asymptotique est de : 1+2+2+h.w.3 *H/h*W /w+ 1 = 6 + 3n = **O(H*W)**

Q11. Le type `np.uint32` *(entier non signé codé sur 32 bits) est-il suffisant pour stocker les éléments de* `S` *si l'image* `A` *comporte 50 millions de pixels ? Justifier.*

L'image `A` est composé d'éléments de type `np.uint8` (soit un octet), la table de sommation ayant chaque élément de son tableau définit comme étant la somme des éléments d'indice strictement inférieur lui, on se met dans le cas le plus défavorable (Si A est composé entièrement d'éléments à une valeur de 255), soit :

$S(H,W) = A.sum = 50.10^6 \times 255 = 1{,}275.10^{10}$

Pour savoir combien de bits sont nécessaires pour coder ce nombre on résout l'inéquation suivante :

$2^n \geq 1{,}275.10^{10} \iff \exp(n.\ln(2)) \geq 1{,}275.10^{10} \iff n.\ln(2) \geq \ln(1{,}275.10^{10}) \iff n \geq \ln(1{,}275.10^{10}) / \ln(2)$ **<=> n ≥ 36,9**

Il faut au moins 37 bits pour stocker les éléments de `S`, si l'image comporte 50 millions de pixels. Le type `np.uint32` n'est donc pas suffisant.

Q12. Écrire une fonction, de complexité O(N), d'entête `def tableSommation(A:image) -> np.ndarray:`

```
def tableSommation(A):
    H, W = A.shape
    S = np.empty((H+1,W+1), np.uint64)
    for i in range(1,H+1):
        for j in range(1,W+1):
            S[i,j] = S[i-1,j] + S[i,j-1] - S[i-1,j-1] + A[i-1,j-1]
    return S
```

Q13. Expliquer en quelques lignes le principe de fonctionnement de `réductionSommation1`.

La fonction `réductionSommation1` prend en compte comme arguments : l'image initiale `A`, la table de sommation `S` et les dimensions de la nouvelle image `w` et `h`, et renvoie l'image finale réduite `a`. Elle segmente l'image initiale en fonction de la taille de l'image finale à l'aide de divisions entières et chaque pixel de l'image finale est la moyenne locale du segment de l'image initiale associé.

Cette moyenne est réalisée à l'aide de la table de sommation qui est manipulée de sorte à obtenir X : la somme des valeurs comprises dans la zone (de `ph*pw` pixel de l'image initiale) à réduire en un pixel de l'image finale. Cette somme est finalement moyennée (en la divisant par `nbp` : le nombre de pixel compris dans cette zone) puis affectée à la nouvelle image réduite.

Q14. Donner sa complexité temporelle asymptotique.

La complexité temporelle asymptotique de la fonction `réductionSommation1` est de :

6 + h.w.(4 + 3) + 1 = 7 + 7n = **O(n) avec : n = h.w**

Q15. Montrer que la fonction `réductionSommation2` *dont le code est fourni ci-dessous donne le même résultat que* `réductionSommation1`.

La fonction `réductionSommation1` utilise une double boucle `for` pour accéder à chaque élément de la matrice `a` (qui est l'image réduite de `A`) et à l'aide d'opérations sur la matrice de sommation `S`, permet d'obtenir la moyenne sur un intervalle de `ph` (sur la hauteur) et `pw` (sur la largeur). On moyenne donc `ph x pw` pixels de l'image originale `A` en un pixel de l'image réduite `a`.

La fonction `réductionSommation2` n'utilise pas de boucle `for`, mais définit une nouvelle matrice `sred`, qui permet d'accéder aux valeurs de la matrice `S` avec un « pas » de `ph` sur les lignes et de `pw` sur les colonnes. Des opérations sur cette matrice `sred`, permettent d'obtenir la valeur moyenne des pixels de l'image originale `A` sur un intervalle de `ph` (sur la hauteur) et de `pw` (sur la largeur) et d'obtenir directement l'image `a`, après avoir pris la valeur arrondie et l'avoir convertie en une valeur sur huit bits.

Q16. Comparer les complexités asymptotiques en temps et en mémoire des deux versions de la fonction `réductionSommation`. *Quel est l'avantage de la seconde version ?*

Soit CAT la complexité asymptotique en temps et CAM la complexité asymptotique en mémoire (qui est également appelée complexité en espace), et les indices 1 et 2 représentant respectivement les fonctions `réductionSommation1` et `réductionSommation2`.

$CAT_1 = O(n)$
$CAT_2 = O(n)$ *(les boucles sont implicites)*
CAM_1 = size(a) + (size(H)+size(W)) + (size(ph)+size(pw)) + size(nbp) + size(X)
= n + 2 + 2 + 1 + n = 2n + 5 = O(n)
CAM_2 = (size(H)+size(W)) + (size(ph)+size(pw)) + size(sred) + size(dc) + size(dl) + size(d)
= 2 + 2 + n + n + n + n = 4n + 4 = O(n)

La seconde version possède une complexité asymptotique temporelle plus faible que celle de la première version (O(1) < O(n)) pour une complexité asymptotique en mémoire équivalente entre les deux versions. L'avantage de la

seconde version est qu'elle est plus rapide que la première (on n'utilise plus une double boucle `for`, mais des opérations sur une matrice de réduction).

Q17. Discuter des cas d'usage respectifs de `procheVoisin`, `moyenneLocale` et `réductionSommation` pour redimensionner une image.

La fonction `procheVoisin` perd de l'information (on ne prend que quelques pixels parmi l'image initiale `A`). La fonction `moyenneLocale` ne fonctionne que si les dimensions de l'image `a` divisent celles de l'image `A`. Finalement, la fonction `réductionSommation` permet de réduire la taille de l'image initiale en prenant en compte l'intégralité de ses éléments, tout fonctionnant même si les dimensions de l'image `a` ne divisent pas celles de l'image `A`.

Q18. Écrire une requête SQL donnant les identifiants de toutes les photographies au ratio 4:3, c'est-à-dire dont le rapport largeur sur hauteur vaut exactement 4/3.

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

Q19. Écrire une requête qui compte le nombre de photos prises par « Alice » ou « Bernard ».

```
SELECT COUNT(*)
FROM Photo
WHERE PH_auteur ='Alice' or  PH_auteur =' 'Bernard' ;
```

Q20. Écrire une requête qui fournit l'identifiant et la date des photographies prises avant 2006 et associées au mot-clé « surf ».

```
SELECT Photo.PH_id, PH_date
FROM Photo
JOIN Decrit
ON Photo.PH_id = Decrit.PH_id
JOIN Motcle
ON Decrit.MC_id = Motcle.MC_id
WHERE EXTRACT(year FROM PH_date) < '2006-01-01' AND MC_texte = 'surf' ;
```

Remarque : au niveau de SELECT, on précise `Photo.PH_id`, car `PH_id` existe dans `Photo` et `Decrit`

Q21. Écrire une requête qui donne le prénom de l'auteur et l'identifiant de tous les selfies, c'est-à-dire les photographies sur lesquelles l'auteur est présent.

```
SELECT PE_Prenom, PH_id
FROM Photo
JOIN Present
ON Photo.PH_id = Present.PH_id
JOIN Personne
ON Photo.PH_auteur = Personne.PE_id ;
```

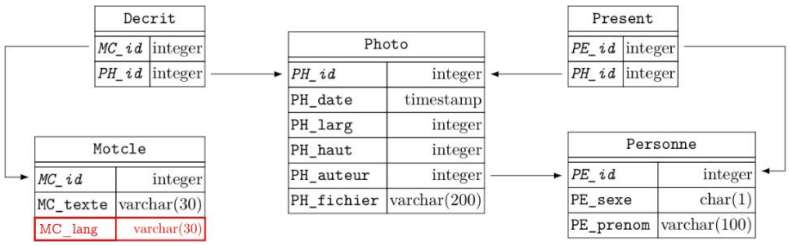
Pour le plaisir 😊 il y avait une dernière question de SQL bien tordue dans le sujet de Centrale : *Écrire une requête qui sélectionne toutes les photographies sur lesquelles sont présents « Alice » et « Bernard », à l'exclusion de toute autre personne.*

```
( SELECT PH_id
FROM Photo
JOIN Present
ON Photo.PH_id = Present.PH_id
JOIN Personne
ON Present.PE_id = Personne.PE_id
WHERE PE_prenom = 'Alice' )
INTERSECT
( SELECT PH_id
FROM Photo
JOIN Present
ON Photo.PH_id = Present.PH_id
```

```
JOIN Personne
ON Present.PE_id = Personne.PE_id
WHERE PE_prenom = 'Bernard' )
EXCEPT
(SELECT PH_id
FROM Photo JOIN Present USING (PH_id)
GROUP BY PH_id HAVING COUNT(*)>2);
```

Q22. Transformation Bdd

On modifie la table Motcle, en ajoutant une caractéristique « langue » : MC_lang



Q23. Requete mountain

```
SELECT Photo.PH_id
FROM Photo
JOIN Decrit
ON Photo.PH_id = Decrit.PH_id
JOIN Motcle
ON Decrit.MC_id = Motcle.MC_id
WHERE MC_lang = 'english' AND MC_texte = 'mountain' ;
```

Q24. Écrire une fonction d'entête `def initMosaïque(source:image, w:int, h:int, p:int) -> image:`

```
def initMosaïque(source, w, h, p):
    H, W = source.shape
    S = tableSommation(source)
    mosaique = réductionSommation2(source, S, w, h)
    return mosaique
```

Q25. Écrire une fonction d'entête `def L1(a:image, b:image) -> int:`

```
def L1(a,b)
    return np.uint64(np.sum(abs(a-b)))
```

Q26. Écrire une fonction d'entête `def choixVignette(pavé:image, vignettes:[image]) -> int:`

```
def choixVignette(pavé, vignettes)
    I = len(vignettes)
    L1_res = np.array(1,I)
    for i in range(0,I)
         L1_res[i]= L1(pavé,vignettes[i])
    return np.argmin(L1_res)
```

Q27. Écrire, à l'aide de ce qui précède, une fonction d'entête `def construireMosaïque(source:image, vignettes:[image], p:int) -> image:`

```
def construireMosaïque(source, vignettes, p)
   h, w = vignettes[0].shape
   m_f = np.zeros((h*p,w*p),np.uint8)
   m_i = initMosaïque(source,w,h,p)
   for i in range(0,h*p,h)
     for j in range(0,w*p,w)
       m_f[i:i+h,j:j+w]= vignettes[choixVignette(m_i[i:i+h,j:j+w],vignettes)]
   return m_f
```

Q28. Déterminer sa complexité temporelle asymptotique en fonction de la taille n = hw des vignettes, du nombre r de vignettes dans la mosaïque et de la longueur q de la liste `vignettes`*.*

La complexité temporelle asymptotique de la fonction `construireMosaïque` est de :

CAT_construireMosaïque = CAT_initMosaïque + O(r) * CAT_choixVignette

or CAT_initMosaïque = CAT_tableSommation + CAT_réductionSommation2,

et CAT_choixVignettes = O(q)

De plus, CAT_tableSommation = O(N) = O(n*r), et CAT_réductionSommation2 = O(1)

Finalement, on a :

CAT_construireMosaïque = O(n*r) + O(1) + O(r*q)

CAT_construireMosaïque = O(r*(n+q))

Q29. Proposer une stratégie de construction de photomosaïque permettant de sélectionner un maximum de vignettes différentes et, au cas où une vignette serait réutilisée, d'éviter que les différentes apparitions de la même vignette se retrouvent trop proches.

Pour sélectionner un maximum de vignettes différentes, on peut définir des groupes de vignettes qui minimisent la fonction L1 qui définit la distance entre deux images, puis lorsque l'on doit affecter une vignette sur la photomosaïque, prendre la vignette minimisant L1 sauf si un des proches voisin possède la même vignette, dans ce cas, piocher dans le groupe de vignettes minimisant L1.

On peut supprimer de la liste la vignette après l'avoir affectée dans la matrice de la mosaïque.

On peut modifier une autre vignette que celle minimisant la fonction L1 (augmenter la transparence, accentuer la luminosité) dans le cas où un des proches voisins possède la même vignette.

Q30. Implanter cette stratégie sous la forme d'une fonction `belleMosaïque`*, version améliorée de la fonction* `construireMosaïque`*, dont on définira les éventuels paramètres supplémentaires.*

L'implémentation la plus simple est de supprimer de la liste des vignettes, la vignette utilisée dans la matrice finale.

```
def belleMosaïque(source, vignettes, p)
    h, w = vignettes[0].shape
    m_f = np.zeros((h*p,w*p),np.uint8)
    m_i = initMosaïque(source,w,h,p)
    for i in range(0,h*p,h)
         for j in range(0,w*p,w)
               i_vignette = choixVignette(m_i[i:i+h,j:j+w],vignettes)
               m_f[i:i+h,j:j+w] = vignettes[i_vignette]
               vignettes = vignettes.pop(i_vignette)
     return m_f
```