Devoir à la maison n°1 -V2 - Corrigé

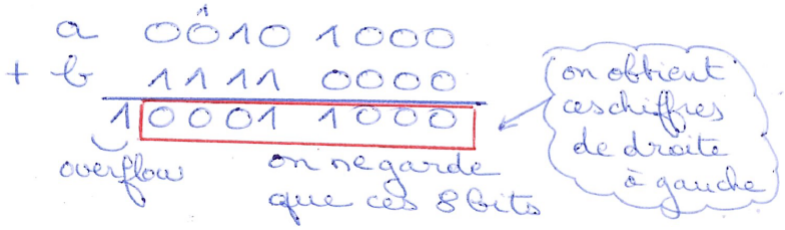
Ce sujet est librement adapté de l'épreuve commune d'informatique de Centrale 2020.

Partie I : Pixel et image

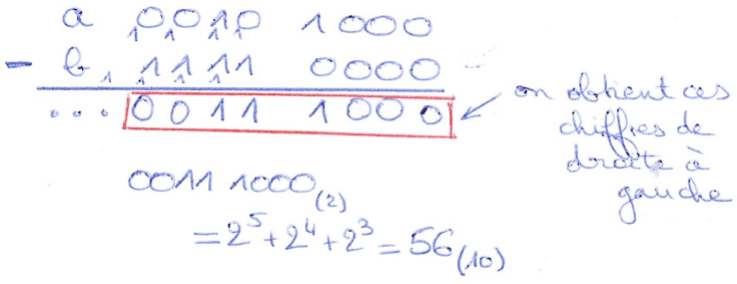
1. Chaque composante RGB d'un pixel étant un entier naturel codé sur 8 bits, elle peut prendre $2^8 = 256$ valeurs différentes : tous les nombres de 0 (codé 00000000) à 255 (codé11111111).
   Puisqu'il y a trois composantes,
   Un pixel dont chaque composante RGB est codée sur 8 bits peut prendre 256^3 couleurs différentes.
2. Le blanc consiste à donner l'intensité maximale aux trois composantes.
   Sur 8 bits, l'intensité maximale est $2^8 - 1 = 255$.

   En pensant à bien spécifier que l'on veut coder les nombres sur un octet seulement, on obtient donc un pixel blanc à l'aide de la commande :

   `np.array([255, 255, 255], np.uint8)`.
3. Le codage binaire du nombre décimal 280 est 100011000.
   En ne conservant que les huit derniers bits, on obtient alors 00011000 qui code bien 24.
4. $$a = 40_{(10)} = 0010 \quad 1000_{(2)}$$
   $$b = 240_{(10)} = 1111 \quad 0000_{(2)}$$
   - Voici le principe de l'addition bit à bit :



   C'est pour cela que l'on obtient $0001 \quad 1000_{(2)} = 24_{(10)}$
   - Voici le principe de la soustraction bit à bit :



   - Le quotient de la division entière de 40 par 240 vaut 0, donc le résultat renvoyé par Python est bien 0.

- Remarquons ici que l'opération de division / retourne toujours un flottant , donc le résultat renvoyé est 0.166.

5. Voici une possibilité de code :

```
def gris(p) #p est un tableau de trois entiers de type np.int8
    return(np.int8(np.sum(p)/3)) #utiliser sum permet d'éviter l'écueil décrit dans la question précédente
```

6. L'expression source.shape renvoie les dimensions du tableau Numpy source. L'image contenue dans le fichier surfer.jpg a donc une hauteur de 3000 pixels, une largeur de 4000 pixels et chaque pixel est représenté par 3 composantes. L'expression `source[0,0]` renvoie les informations du pixel en haut à gauche de l'image, qui est donc un pixel avec $144/255 \approx 56\%$ de rouge, $191/255 \approx 75\%$ de vert et $221/255 \approx 87\%$ de bleu. *(les pourcentages n'étaient pas attendus des candidats)*

7. Voici une possibilité de code :

```
def conversion(a):
    m, n, p = a.shape #p sera égal à 3 si le tableau a en entrée est correct
    b = np.zeros([m,n], np.uint8) #b sera le tableau en niveau de gris ,
                                  #au début tous les pixels sont à zéros
    for i in range(m):
        for j in range(n):
            b[i, j] = gris(a[i, j])
    return(b)
```

Partie II : Redimensionnement d'images

8. La photomosaïque fait 2m de large, soit 2000mm, soit 20000 pixels en largeur. Comme il y aura 40 vignettes sur la largeur, cela fait 500 pixels de large par vignettes.
   Puisque chaque vignette est au format $4:3$, elles feront $\frac{3}{4} \times 500 = 375$ pixels de haut.

   La taille de chaque vignette est donc 500×375 pixels.

   La photomosaïque fera donc au total 20000×15000 pixels, soit 300 Mégapixels.

II.B : Algorithme au plus proche voisin

9. Voici une possibilité de code :

```
def proche_voisin(A, w, h):
    H,W = A.shape
    res = np.zeros((h,w), np.uint8)
    for i in range(h):
        for j in range(w):
            i0 = i*H//h
            j0 = j*W//w
            res[i, j] = A[i0, j0]
    return res
```

10. On peut considérer que la complexité de l'instruction `np.shape` est constante, et que la complexité de `np.zeros` utilisée est en $O(wh)$.

    Chaque passage dans les 2 boucles `for` imbriquées réalise un nombre constant d'opérations élémentaires, et il y a $w \times h$ passages, d'où une complexité en $O(wh)$ pour ces boucles.

    On a donc une complexité finale de $O(wh)$.

II.C : Algorithme de réduction par moyenne locale

11. Il s'agit d'expliquer ce programme :

```
1  def  moyenneLocale(A:image, w:int, h:int) -> image:
2      a = np.empty((h, w), np.uint8)
3      H, W = A.shape
4      ph, pw =  H // h, W // w
5      for I in range(0, H, ph):
6          for J in range(0, W, pw):
7              a[I // ph, J // pw] = round(np.mean(A[I:I+ph, J:J+pw]))
8      return a
```

Chaque pixel de la nouvelle image récupère comme valeur **la moyenne des valeurs dans un rectangle de dimensions `ph` et `pw`** situé au bon endroit. Pour cela, `I//ph` et `J//pw` parcourent les coordonnées de l'image d'arrivée, et la tranche `[I:I+ph, J:J+pw]` désigne donc des rectangles formant une partition de l'image.

12. On peut considérer que la complexité de l'instruction `np.shape` est constante, et que la complexité de `np.empty` utilisée est en $O(wh)$.

La première boucle `for` tourne $\frac{H}{ph} = h$ fois (on a pris en compte le pas de la boucle), et la deuxième `for` tourne $\frac{W}{pw} = w$ fois. **A l'intérieur de ces 2 boucles imbriquées**, chaque appel à la fonction `np.mean()` utilise $O(ph \times pw)$ opérations.

Ce qui fait que les boucles ont un coup en $O(h * w * ph * pw) = O(HW)$.

Finalement on a une complexité totale en $O(H \times W)$.

II.D : Optimisation de la réduction par moyenne locale

13. Voici le tableau de sommation complété :

2	3	2
10	7	1
4	5	11

tableau A

0	0	0	0
0	2	5	**7**
0	12	22	**25**
0	16	**31**	**45**

tableau S

FIGURE 1 – tableauSommation

14. La question est de savoir si une somme de 50 millions de valeurs entre 0 et 255 peut dépasser $2^{32} - 1$.
Dans le pire des cas (si tous les pixels sont blancs) la valeur maximale à stocker dans `S` est 255×50.10^6.

Nous devons donc comparer $x = 255 \times 50.10^6$ avec $y = 2^{32} - 1$.
x est environ $2^8.50.10^6$, y est environ 2^{32}, donc cela revient à comparer 50.10^6 avec 2^{24}.
Rappelons que $2^{10} = 1024$ environ égal à 10^3, donc finalement on compare 50 à $2^4 = 16$.

Finalement, x est plus grand que y, donc non un stockage sur 32 bits n'est pas suffisant.

15. **Méthode 1 :**

```
1  def table_sommation(A):
2      H,W = A.shape
3      res  = np.zeros((H+1, W+1), np.uint64)
4      for  i  in  range(1,  H+1):
5          for  j  in  range(1,  W+1):
6              res[i ,  j]  = res[i-1, j]  + res[i ,  j-1] - res[i-1, j-1]    +    A[i-1, j-1]
7      return  res
```

On utilise une astuce ici : $\sum_{\substack{0\leq i<l\\0\leq j<c}} A(i,j) = \sum_{\substack{0\leq i<l-1\\0\leq j<c}} A(i,j) + \sum_{\substack{0\leq i<l\\0\leq j<c-1}} A(i,j) - \sum_{\substack{0\leq i<l-1\\0\leq j<c-1}} A(i,j) + A(l,c).$

Cette astuce permet d'assurer une complexité totale en $O(HW) = O(N)$ car nous avons deux boucles imbriquées de coup constant.

Méthode 2 : En utilisant une variable qui stocke la somme des coefficients de chaque ligne au fur et à mesure

```
1  def tableSommation(A):
2      H,W=np.shape(A)
3      S=np.zeros([H+1,W+1])
4      for  i  in  range(1,H+1):
5          som_ligne=A[i-1,0]
6          for  j  in  range(1,W+1):
7
8              S[i ,j]=som_ligne+S[i-1,j]
9              if  j!=W:
10                 som_ligne+=A[i-1,j]
11     return(S)
```

Cette fonction admet aussi une complexité en $O(HW) = O(N)$ car nous avons aussi deux boucles imbriquées de coup constant.

16. Il s'agit d'expliquer ce programme :

```
1  def réductionSommation(A:image, S:np.ndarray, w:int ,  h: int ) -> image:
2      a = np.empty((h, w), np.uint8)
3      H, W = A.shape
4      ph,  pw =  H // h, W // w
5      nbp = ph * pw
6      for  I  in  range(0,  H,  ph):
7          for  J  in  range(0,  W,  pw):
8              X = (S[I+ph, J+pw] - S[I+ph, J]) - (S[I, J+pw] - S[I, J])
9              a[I  // ph,  J  // pw] = round(X / nbp)
10     return  a
```

Cette fonction utilise S pour calculer les sommes sur chaque rectangle de dimension $ph \times pw$, à l'aide de la formule de la ligne 8 (formule assez similaire à celle utilisée dans la question précédente).

Cette somme est donc calculée en $O(1)$ à la ligne 8 pour chaque petit rectangle, et mise à la bonne place dans la matrice `a` à la ligne 9, après une division par le nombre de cases et un arrondi.

C'est le même procédé de réduction d'image que la fonction **`moyenneLocale`**, la différence est que l'utilisation de la table de sommation nous évite d'utiliser la fonction **`np.mean`**.

17. La première boucle tourne $\frac{H}{ph} = h$ fois, et la deuxième **`for`** tourne $\frac{W}{pw} = w$ fois.

Les lignes 8 et 9 s'exécutant en temps borné, on peut affirmer que $\boxed{\text{la complexité temporelle est de } O(h \times w)}$,

II.E : Synthèse

18. 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**).

    Les programmes **moyenneLocale** et **réductionSommation** ont la même complexité totale en $O(HW)$ (dimensions de l'image d'origine). Cependant, une fois le tableau de sommation calculé, chaque redimensionnement suivant est en $O(hw)$ en utilisant **réductionSommation**. C'est donc la méthode à privilégier si on pense effectuer plusieurs redimensionnement d'une même image à plusieurs échelles différentes.

Partie IV : Placement des vignettes

19. Voici une possibilité de code, utilisant l'algorithme du plus proche voisin pour redimensionner les photos :

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

20. Voici une possibilité de code :

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

    Remarque : on n'utilise pas **abs** *car dans le cas où la différence est négative, le codage sur un octet conduit à une erreur. Cela a été expliqué dans la première partie du problème.*

21. Voici une possibilité de code :

```
def choix_vignette(pave, vignettes) :
    i_min = 0
    d_min = L1(pave, vignettes[0]) # initialisation de la distance minimum
    for i in range(1, len(vignettes)):
        distance = L1(pave, vignettes[i])
        if distance < d_min:
            d_min = distance
            i_min = i
    return i_min
```

22. Voici une possibilité de code :

```
def construire_mosaique(source, vignettes, p):
    w,h = np.shape(vignettes[0]) #les dimensions d'une vignette
    res = initMosaique(source, w, h, p) #on initialise le résultat aux bonnes dimensions
    for i in range(p): #on va parcourir l'image bloc par bloc
        for j in range(p):
            bloc = res[i*h:(i+1)*h, j*w:(j+1)*w]
            indice = choix_vignette(bloc, vignettes) #on choisit la vignette la plus proche du bloc
            res[i*h:(i+1)*h, j*w:(j+1)*w] = vignettes[indice] #on remplace le bloc par la bonne vignette
    return res
```

23. L'appel à la fonction **initMosaique** se fait avec une complexité égale à celle de **procheVoisin**, qui est donc en $O(wp \times hp) = O(nr)$ (car $r = p^2$ le nombre total de vignettes)

Ensuite les deux boucles imbriquées s'exécutent $p \times p = r$ fois.
A chaque tour, le choix des vignettes s'exécute en $O(qn)$ (q vignettes à parcourir, et à chaque fois l'appel à `L1` est en $O(n)$).

Au final la complexité de cette fonction est en $O(nr) + rO(qn) = O(nr) + O(qnr)$, soit $\boxed{\text{en } O(qnr)}$.

24. Pour cette question et la suivante, il n'y a pas vraiment de bonne stratégie, mais voici quelques pistes :
    - On peut retirer une vignette déjà utilisée par un voisin quand on choisit les vignettes. On peut même donner un "rayon" dans lequel une vignette ne sera plus utilisable.
    - Si on fait ça, l'ordre de calcul des vignettes se met à avoir de l'importance, et si on ne fait pas attention, on risque de voir des motifs répétés dans des zones uniformes (la même succession de vignettes).
    - On peut par exemple attribuer les vignettes dans un ordre aléatoire, en ignorant, quand on choisit une vignette, celles qui sont déjà attribuées à des pavés voisins.
    - Dans ce dernier cas, la structure la plus simple à utiliser, c'est une matrice $p \times p$, qui contient dans chaque case l'indice de la vignette utilisée (initialisée à -1 partout, par exemple, quand la vignette n'est pas encore choisie). Quand on veut déterminer un choix pour une vignette, on commence par regarder les numéros des vignettes autour, on les met dans une liste, et au moment de choisir une vignette, on commence par exclure ceux de cette liste.

25. Voici une possibilité de code, qui utilise l'implémentation la plus simple consistant à supprimer de la liste des vignettes, la vignette utilisée dans la matrice finale :

```
1  def bellemosaique (source, vignettes, p)
2      h,w= vignettes [0]. shape
3      im_finale = np.empty ((h*p,w*p),np.unint8)
4      im_initiale =initmosaique (source, w,h,p )
5      for i in range (0,h*p,h)
6          for j in range (0,w*p,w)
7              i_vignette = choixVignette ( im_initiale (i:i+h,j:j+w), vignettes )
8              im_finale [i:i+h,j:j+w]=vignettes[i_vignette]
9              vignettes = vignettes.pop( i_vignette )
10     return im_finale
```