Photomosaïque

I Pixels et images

I.1 Pixels

Q1 Un pixel peut être représenté par une séquence de 3 entiers de 8 bits chacun.
Chaque entier appartient ainsi à l'intervalle $[\![0; 2^8 - 1]\!]$ et peut donc prendre 256 valeurs.
Les valeurs prises par chaque entier sont indépendantes l'une de l'autre, donc il y $256^3 = 2^{24}$ tuples possibles.
On peut donc représenter 2^{24}, i.e. 16 777 216 couleurs, avec un tel pixel.

Q2 Un pixel blanc peut être créé par l'instruction :

```
1 np.array([255, 255, 255], np.uint8) # ou np.full(3, 255, np.uint8)
```

Q3 Soit `a = np.uint8(280)` et `b = np.uint8(240)`.
Les variables **a** et **b** sont de type `uint8`, donc leurs valeurs doivent être entendues modulo 256 (car $256 = 2^8$). Le résultat du calcul est de même type (donc modulo 256). [1] Ainsi,

- `a` vaut $280 - 256$, càd. `a` vaut 24,
- `b` vaut 240,
- `a+b` vaut $240 + 24 - 256$, càd. `a+b` vaut 8,
- `a-b` vaut $24 - 240 + 256$, càd. `a-b` vaut 40,
- `a//b` est le quotient entier de la division de 24 par 240, càd. `a//b` vaut 0,
- `a/b` est le quotient décimal de la division de 24 par 240, càd. `a/b` vaut 0.1.

Q4 La moyenne fait une division et renvoie un flottant (type `np.float64`, i.e. `float`).
La fonction `round` permet d'arrondir au plus proche entier (comme précisé dans l'annexe), au type `int`. Or, l'énoncé demande un type `np.uint8`, donc un changement de type.

```
10 def gris(p:pixel) -> np.uint8:
11     return np.uint8(round(np.mean(p)))
12 # ou np.uint8(round(np.sum(p)/3))
```

I.2 Images

Q5 Après la commande `source = plt.imread("surfer.jpg")`,

- `source.shape` renvoie `(3000, 4000, 3)`.
  Ceci indique que l'image a 3 dimensions : 3 000 lignes, 4 000 colonnes et, à l'intersection de chaque ligne et colonne, une liste à 3 éléments.
- `source[0,0]` renvoie `np.array([144, 191, 221], np.uint8)`.
  Ceci indique que le pixel en haut à gauche de l'image (en `[0,0]`) a 3 composantes de couleurs, respectivement Red, Green et Blue, qui valent respectivement 144, 191 et 221 [2].

Q6

```
17 def conversion(a:np.ndarray) -> image:
18     h, w = a.shape[0], a.shape[1] # h,w,p = a.shape est aussi possible
19     img = np.zeros((h,w), np.uint8) # ou img = np.empty((h,w), np.uint8)
20     for i in range(h):
21         for j in range(w):
22             img[i, j] = gris(a[i, j])
23     return img
```

1. Par contre, `np.sum(t)` d'un tableau `t` à deux entiers `uint8` change le type en `uint32`...
2. Il s'agit d'un pixel bleu ciel, comme constaté sur l'image en figure 1.

II Redimensionnement d'images

II.1 Le contexte

Q7 Si $W = 2$ m $= 2\,000$ mm, on peut y disposer 40 vignettes si et seulement si $w = 50$ mm (car $40 \times 50 = 2\,000$).
Comme $h = \frac{3}{4}w$, il s'ensuit que $h = \frac{3}{4} \times 50$. Donc, $h = 37{,}5$ mm.
Si la résolution est de 10 pixels par millimètre, la taille de vignette $w \times h$, en pixels, est $\boxed{500 \times 375}$.

Chaque vignette est donc constituée de $500 \times 375 = 187\,500$ pixels.
Il y a $40 \times 40 = 1\,600$ vignettes dans la photomosaïque.
La taille de la photomosaïque est donc de $187\,500 \times 1\,600$ pixels, càd. $\boxed{300 \text{ millions de pixels}}$.

II.2 Algorithme d'interpolation au plus proche voisin

Q8 On applique la formule $a(i,j) = A(\lfloor \frac{iH}{h} \rfloor, \lfloor \frac{iW}{w} \rfloor)$ à chaque couple (i,j) de $[\![0, h-1]\!] \times [\![0, w-1]\!]$.
Par construction, l'image va être fortement crénelée.

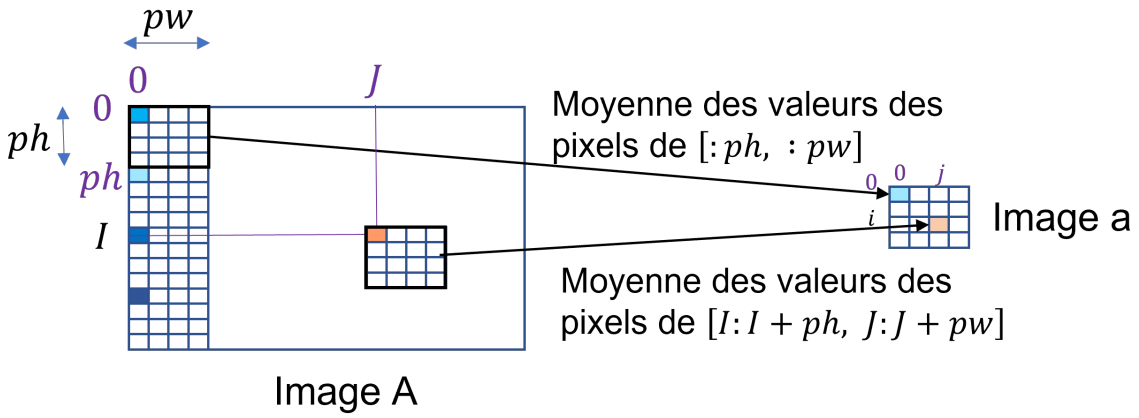
```
28 def procheVoisin(A:image, w:int, h:int) -> image:
29     H, W = A.shape # A est en niveau de gris, donc de dimension 2
30     a = np.zeros((h,w), np.uint8)
31     for i in range(h):
32         for j in range(w):
33             a[i, j] = A[i*H//h, j*W//w] # ou A[math.floor(i*H/h), math.
   floor(j*W/w)]
34     return a
```

Q9 À l'extérieur des boucles imbriquées, les instructions sont en $\mathcal{O}(1)$.
Pour chacun des hw couples (i,j), on exécute une instruction en $\mathcal{O}(1)$: la complexité temporelle asymptotique des boucles imbriquées est donc en $\mathcal{O}(hw) = \mathcal{O}(n)$ avec $n = hw$.
Par somme, la complexité temporelle asymptotique de `procheVoisin` est en $\boxed{\mathcal{O}(n)}$.

II.3 Algorithme de réduction par moyenne locale

On suppose dans cette partie que H **et** W **sont respectivement des multiples de** h **et** w.

Q10 Illustrons ce que fait la fonction `moyenneLocale`.



La fonction `moyenneLocale` attribue au pixel $(i,j) = (\lfloor \frac{Ih}{H} \rfloor, \lfloor \frac{Iw}{W} \rfloor)$ de l'image `a` la meilleure approximation entière de la moyenne des pixels du rectangle `A[I:I+ph, J+J+pw]` .

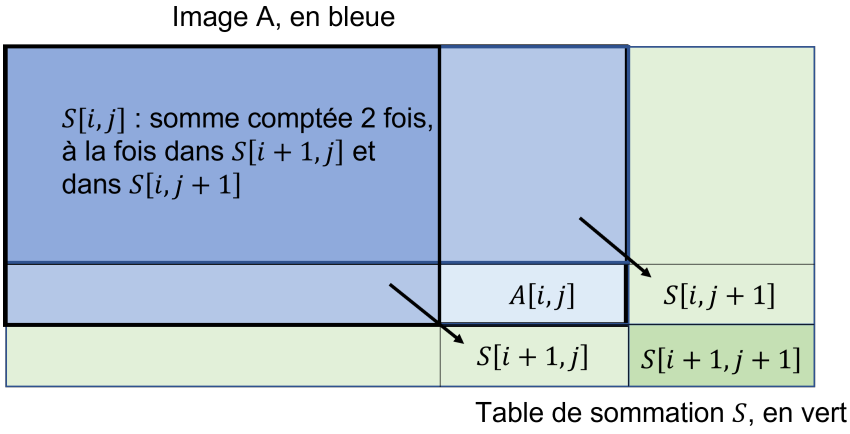
Q11 À l'extérieur des boucles imbriquées, les instructions sont en $\mathcal{O}(1)$.
Comme la boucle en I a pour pas ph et celle en J a pour pas pw, on exécute $\frac{H}{ph} \times \frac{W}{pw}$ fois les instructions de la boucle imbriquée.
Pour chacun des couples (I, J), on exécute `np.mean(A[I:I+ph, J:J+pw])` qui parcourt $ph \times pw$ valeurs de `A` : la complexité temporelle asymptotique des boucles imbriquées est donc en $\mathcal{O}(\frac{H}{ph} \times \frac{W}{pw} \times ph \times pw) = \mathcal{O}(HW) = \mathcal{O}(N)$ avec $N = HW$.
Par somme, la complexité temporelle asymptotique de `moyenneLocale` est en $\boxed{\mathcal{O}(N)}$.

II.4 Optimisation de la réduction par moyenne locale

Q12 Le calcul de `S[H+1, W+1]` (pire des cas) nécessite de calculer la somme des valeurs de 50 millions de pixels. Si chacune de ces valeurs vaut 255 (plus grande valeur d'un entier de type `uint8`), alors la somme vaut $255 \times 50 \times 10^6 = 12\ 750\ 000\ 000$.
Or, la plus grande valeur prise par un entier de type `uint32` est $2^{32} - 1 = 4\ 294\ 967\ 295$.
Par conséquent, $\boxed{\text{le type } \texttt{uint32} \text{ est insuffisant pour stocker les éléments de } \texttt{S}}$.

Q13 Notons que $2^{64} - 1 = 18\ 446\ 744\ 073\ 709\ 551\ 615$ est largement supérieur à la plus grande valeur prise par `S`. Nous utilisons donc ce type pour les éléments de `S` dans la fonction `tableSommation` qui suit.

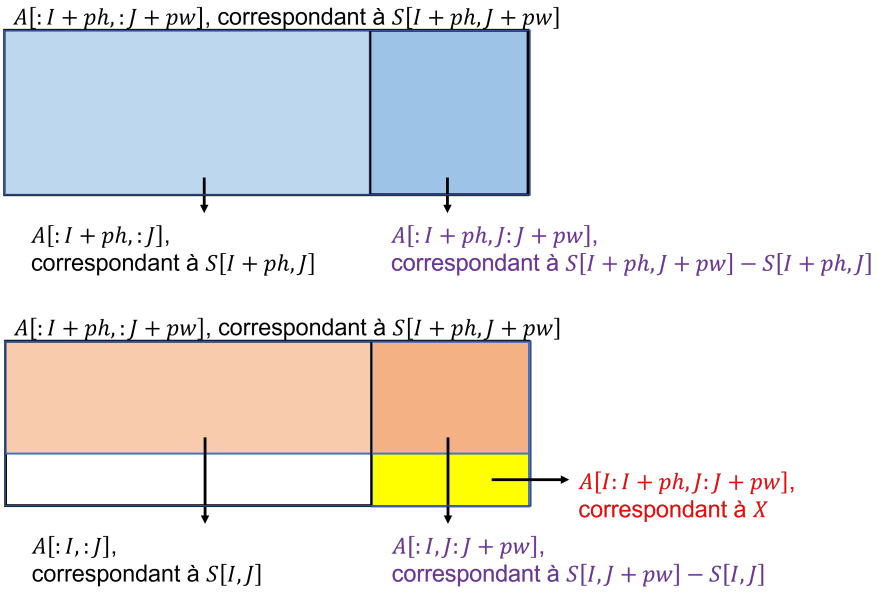
La complexité temporelle asymptotique attendue est en $\mathcal{O}(N)$, donc `A` doit être parcouru un nombre fini de fois. On propose de calculer, pour tout $i \in [\![0, H-1]\!]$ et tout $j \in [\![0, W-1]\!]$, les valeurs de `S` par `S[`$i+1, j+1$`] = S[`$i+1, j$`] + S[`$i, j+1$`] - S[`i, j`] + A[`i, j`]`, comme illustré dans la figure suivante :



D'où le code de la fonction `tableSommation`.

```
45 def tableSommation(A:image) -> np.ndarray:
46     H, W = A.shape
47     S = np.zeros((H+1,W+1), np.uint64) # valeurs par défaut à zéro
48     for i in range(H):
49         for j in range(W):
50             S[i+1, j+1] = S[i+1, j] + S[i, j+1] - S[i, j] + A[i, j]
51     return S
```

Q14 Comme on peut le constater sur la figure ci-dessous, la valeur de `X`, ligne 8 de la fonction `réductionSommation1`, correspond à la somme des éléments de `A[I:I+ph, J:J+pw]`.



L'instruction `round(X/nbp)`, de la ligne 9, donne alors la meilleure approximation entière de la moyenne des valeurs des pixels du rectangle `A[I:I+ph, J:J+pw]`.
Ainsi, la fonction `réductionSommation1` attribue au pixel $(i,j) = (\lfloor \frac{Ih}{H} \rfloor, \lfloor \frac{Iw}{W} \rfloor)$ de l'image `a` la meilleure approximation entière de la moyenne des pixels du rectangle `A[I:I+ph, J:J+pw]`.
Il s'agit donc d'un algorithme qui réalise fonctionnellement la même chose que la fonction `moyenneLocale`.

Q15 On reprend le raisonnement de **Q11**.
La différence est que `S` est donné en paramètre à la fonction `réductionSommation1`.
Ainsi, pour chacun des $\frac{H}{ph} \times \frac{W}{pw}$ couples (I,J), on exécute des instructions en $\mathcal{O}(1)$:
la complexité temporelle asymptotique des boucles imbriquées est donc en
$\mathcal{O}(\frac{H}{ph} \times \frac{W}{pw} \times 1) = \mathcal{O}(hw) = \mathcal{O}(n)$ avec $n = hw$.
Par suite, la complexité temporelle asymptotique de `réductionSommation1` est en $\boxed{\mathcal{O}(n)}$.

Q16 Décrivons les instructions de la fonction `réductionSommation2` :

- Les lignes 1 et 2 servent à l'initialisation des variables `H, W, ph` et `pw`.
- Ligne 4 : l'instruction `sred = S[0:H+1:ph, 0:W+1:pw]` initialise `sred` avec les valeurs de la vue sur `S` tel que, à tout $(i,j) \in [\![0,h]\!]$, on ait `sred`(i,j) =`S`(I,J) avec $I = i \times ph$ et $J = j \times pw$.
- Ligne 5 : l'instruction `dc = sred[:, 1:] - sred[:, :-1]` initialise `dc` avec les valeurs de la vue sur `sred` tel que, à tout $(i,j) \in [\![0,h]\!]$, on ait :
`dc`(i,j) =`sred`$(i,j+1)-$`sred`(i,j) = `S`$(I,J+pw)-$`S`(I,J).
- Ligne 6 : l'instruction `d1 = dc[1:, :] - dc[:-1, :-]` initialise `d1` avec les valeurs de la vue sur `dc` tel que, à tout $(i,j) \in [\![0,h]\!]$, on ait `d1`(i,j) =`dc`$(i+1,j)-$ `dc`(i,j).
Ainsi, `d1`(i,j) = (`S`$(I+ph,J+pw)-$`S`$(I+ph,J))$ $-$ (`S`$(I,J+pw)-$`S`(I,J)) : il s'agit donc de la somme des valeurs des pixels du rectangle `A[I:I+ph, J:J+pw]`, comme vu en **Q14**.
- Ligne 7 : l'instruction `d = d1/ (ph * pw)` initialise `d` avec les valeurs de la vue sur `d1` tel que, à tout $(i,j) \in [\![0,h]\!]$, `d`(i,j) vaille la moyenne des valeurs des pixels du rectangle `A[I:I+ph, J:J+pw]`.

- Ligne 8 : l'instruction `return np.uint8(d.round())` renvoie un tableau, appelons-le `a`, tel que, pour tout $(i,j) \in [\![0,h]\!]$, `a`(i,j) contienne la meilleure approximation entière de la moyenne des pixels du rectangle `A[I:I+ph, J:J+pw]`, de type `uint8`.

Ainsi, la fonction `réductionSommation2` donne le même résultat que `réductionSommation1`.

Q17 Complexité temporelle asymptotique de la fonction `réductionSommation2` :

- Les lignes 2 et 3 sont en $\mathcal{O}(1)$.
- la ligne 4 sélectionne $hw = n$ éléments de `S` et les affectent à `sred` : on peut estimer que `numpy` optimise la sélection avec une complexité en $\mathcal{O}(n)$. L'alimentation de `sred` a lieu en $\mathcal{O}(n)$.
- Les instructions des lignes 5 à 8 ont lieu sur des tableaux de taille n, donc avec une complexité en $\mathcal{O}(n)$.

Ainsi, la complexité temporelle asymptotique de `réductionSommation2` est en $\boxed{\mathcal{O}(n)}$.

Les fonctions `réductionSommation1` et `réductionSommation2` ont donc la même complexité temporelle asymptotique.
Cependant, `réductionSommation2` utilise des primitives de `numpy` pour la gestion des tableaux, dont le code écrit en C est notoirement optimisé. On peut donc supposer que le coefficient multiplicatif devant n est plus faible pour cette seconde version.

Les fonctions `réductionSommation1` et `réductionSommation2` ont les mêmes paramètres et utilisent des tableaux locaux de taille n : elles ont donc la même complexité en mémoire. Cependant, la fonction `réductionSommation2` utilise plusieurs tableaux intermédiaires (`sred`, `dc`, `d1`, `d`) de taille n, alors que la fonction `réductionSommation1` n'en utilise qu'un seul (`a`). La complexité asymptotique en mémoire de la fonction `réductionSommation1` est ainsi la même que celle de fonction `réductionSommation2`, mais avec un meilleur coefficient multiplicatif.

Pour conclure, le gain espéré de performance en temps de la fonction `réductionSommation2` sera partiellement contrebalancé par la perte de performance en mémoire.

II.5 Synthèse

Q18 La fonction `procheVoisin` fournit un premier redimensionnement de l'image source en $\mathcal{O}(n)$, mais le résultat crénelé, même s'il est rapidement obtenu, ne sera pas forcément perçu comme satisfaisant.

La fonction `moyenneLocale` fournit un premier redimensionnement de l'image source en $\mathcal{O}(N)$. Le résultat sera perçu de meilleure qualité que pour la fonction précédente, mais il est plus long à produire.

Les fonctions `réductionSommation` nécessitent un prétraitement en $\mathcal{O}(N)$, puis s'exécutent en $\mathcal{O}(n)$. Le résultat est le même que celui de `moyenneLocale`, le temps d'exécution étant similaire. Cependant, les fonctions `réductionSommation` sont à privilégier si on souhaite faire plusieurs fois la réduction de la même image, car la réduction sera à chaque fois en $\mathcal{O}(n)$.

Il reste à soulever le fait que les fonctions `moyenneLocale` et `réductionSommation` nécessitent que $\frac{H}{h}$ et $\frac{W}{w}$ soient entiers. Cette contrainte sur le facteur de réduction limite de facto leurs possibilités d'utilisation.

III Sélection des images de la banque

III.1 Quelques requêtes (SQL)

On privilégie une rédaction des requêtes qui utilise les possibilités du langage SQL, telles que données en annexe de l'énoncé.

Q19 La division décimale ne donnant pas forcément de résultat exact, on doit utiliser la multiplication :
SELECT PH_id FROM Photo WHERE PH_larg * 3 = PH_haut * 4

Q20
```
SELECT COUNT(PH_id)
FROM Photo
JOIN Personne ON PH_auteur = PE_id
WHERE PE_prenom = 'Alice' OR PE_prenom = 'Bernard'
```

Q21
```
SELECT PH_id, PH_date
FROM Motcle
JOIN Decrit USING MC_id
JOIN Photo USING PH_id
WHERE EXTRACT(year FROM PH_date) < '2006' AND MC_texte = 'surf'
```

Q22 On propose un affichage par prénom, afin d'avoir les selfies par prénom.
```
SELECT PE_prenom, PH_id
FROM Photo
JOIN Present ON PH_auteur = PE_id
JOIN Personne USING PE_id
ORDER BY PE_prenom
```

Q23 On comprend que l'on demande les photographies sur lesquelles se trouvent **à la fois** Alice et Bernard (sans cela l'énoncé aurait écrit Alice ou Bernard), à l'exclusion de toute autre personne.
On propose de sélectionner les photos où se trouve à la fois Alice et Bernard et dont le total des personnes présentes sur une photo est exactement égal à 2.
```
SELECT PH_id
FROM Present
JOIN Personne USING PE_id
WHERE PE_prenom = 'Alice'

INTERSECT

SELECT PH_id
FROM Present
JOIN Personne USING PE_id
WHERE PE_prenom = 'Bernard'

INTERSECT

SELECT PH_id
FROM Present
GROUP BY PH_id
HAVING COUNT(PH_id) = 2
```

III.2 Internationalisation des mots-clés

Q24 Conceptuellement, à chaque identifiant de mot clé (MC_id), on va associer 1 à n langues (attribut MC_langue, code ISO sur 3 caractères fixes) et un texte associé (attribut MC_texte, pouvant contenir un texte en langue étrangère).

Si l'on traduit cela sans précaution dans le modèle physique de données, on sera amené à créer une nouvelle table, en plus de la table `Motcle`, cette dernière ne conservant qu'une seule colonne avec MC_id. La table `Motcle` ne sera alors plus utilisée, les jointures se faisant toujours avec la nouvelle table créée qui contient aussi MC_id.

C'est pourquoi, on préfère une dénormalisation du modèle conceptuel de données en modifiant la table `Motcle`, en lui ajoutant une colonne `MC_langue`. La clé primaire de `Motcle` sera alors le couple (MC_id, MC_langue) :

Motcle	
MC_id	integer
MC_langue	char(3)
MC_texte	varchar(30)

Q25 Dans la requête qui suit, on n'est pas obligé de sélectionner la langue ET le mot-clé si l'on est sûr que ce dernier est unique dans la table `Motcle`.

SELECT PH_id
FROM Decrit
JOIN Motcle USING MC_id
WHERE MC_langue = 'ENG' AND MC_texte = 'mountain'

IV Placement des vignettes

IV.1 Préparatifs

Q26 On veut créer une image (la photomosaïque) de hauteur $p \times h$ et de largeur $p \times w$: elle contient alors p^2 vignettes de dimension $h \times w$.

Par ailleurs, elle est homothétique de l'image source. A priori, la photomosaïque est plus grande que l'image source, donc les rapports $\frac{H}{h}$ et $\frac{W}{w}$ ne sont pas entiers. Par ailleurs, on n'a pas besoin d'une grande similarité avec toute l'image source, mais uniquement aux pixels multiples de $\lfloor \frac{H}{p \times h} \rfloor$ de l'image source.

C'est pourquoi, on utilise la fonction `procheVoisin`.

```
78 def initMosaique(source:image, w:int, h:int, p:int) -> image:
79     return procheVoisin(source, w*p, h*p)
```

Q27 Afin de prendre en compte les dépassements de capacité éventuels qui fausseraient le résultat, on traduit chacune des valeurs des tableaux `a` et `b`, de type `uint8`, en `int`. La longueur obtenue sera alors significative, et bien de type `int`.

```
82 def L1(a:image, b:image) -> int:
83     l = 0
84     for i in range(h):
85         for j in range(w):
86             s += abs(int(a[i,j]) - int(b[i,j]))
87     return s
```

Q28

```
90  def choixVignette(pavé:image, vignettes:[image]) -> image:
91      m = L1(pavé, vignettes[0])
92      i_m = 0
93      for i in range(len(vignettes)):
94          dis = L1(pavé, vignettes[i])
95          if dis < m:
96              m = dis
97              i_m = i
98      return i_m
```

IV.2 Méthode sans restriction du choix des vignettes

Q29

```
101 def construireMosaique(source:image, vignettes:[image], p:int) -> image:
102     h, w = vignettes[0].shape
103     img_mosaique = initMosaique(source, w, h, p)
104     for I in range(0, p*h, h):
105         for J in range(0, p*w, w):
106             pavé = img_mosaique[I:I+h, J:J+w]
107             ind = choixVignette(pavé, vignettes)
108             img_mosaique[I:I+h, J:J+w] = vignettes[ind]
109     return img_mosaique
110
111 # ou, par changement d'indice
112 def construireMosaique(source:image, vignettes:[image], p:int) -> image:
113     h, w = vignettes[0].shape
114     img_mosaique = initMosaique(source, w, h, p)
115     for i in range(p):
116         for j in range(p):
117             pavé = img_mosaique[i*h:(i+1)*h, j*w:(j+1)*w]
118             ind = choixVignette(pavé, vignettes)
119             img_mosaique[i*h:(i+1)*h, j*w:(j+1)*w] = vignettes[ind]
120     return img_mosaique
```

Q30 Complexité de la fonction `ConstruireMosaique` :
On détermine la complexité temporelle asymptotique à partir de la version de la fonction obtenue après changement d'indice.

- `h, w = vignettes[0].shape` est en $\mathcal{O}(1)$.
- `img_mosaique = initMosaique(source, w, h, p)` correspond à `procheVoisin(source, w*p, h*p)`, de complexité $\mathcal{O}(w \times p \times h \times p) = \mathcal{O}(nr)$ avec $n = hw$ et $r = p^2$.
- Pour chacun des $p^2 = r$ couples (i, j) :
  - `pavé = img_mosaique[I:I+h, J:J+w]` a pour complexité $\mathcal{O}(h \times w) = \mathcal{O}(n)$, car l'affectation de `img_mosaique[I:I+h, J:J+w]` à `pavé` parcourt toutes ses cellules.
  - `ind = choixVignette(pavé, vignettes)` a pour complexité $\mathcal{O}(qL)$ avec q, longueur de `vignettes`, et L, la complexité de `L1(pavé, vignettes[i])`. Or, cette dernière fonction a pour complexité $\mathcal{O}(h \times w) = \mathcal{O}(n)$.
    Donc, `ind = choixVignette(pavé, vignettes)` a pour complexité $\mathcal{O}(nq)$.
  - `img_mosaique[I:I+h, J:J+w] = vignettes[ind]` a pour complexité $\mathcal{O}(h \times w) = \mathcal{O}(n)$, car l'affectation à `img_mosaique[I:I+h, J:J+w]` parcourt toutes ses cellules.

La complexité temporelle asymptotique de la boucle est donc en $\mathcal{O}(nr(q+2)) = \mathcal{O}(nrq)$.

Ainsi, la complexité temporelle asymptotique de la fonction `ConstruireMosaique` est en $\mathcal{O}(1 + nr(1 + q))$, càd. $\boxed{\mathcal{O}(nrq)}$.

IV.3 Améliorations

Q31 On propose la stratégie de construction suivante.

Si une image a déjà été utilisée, on la marque et on utilise un deuxième choix (en terme de proximité définie par `L1`). Si le deuxième choix a déjà été utilisé, on utilise un troisième choix et ainsi de suite.
Si l'on a tenté d'utiliser une même image, par exemple 10 fois, et que l'on utilisé d'autres images à la place (cf. ce qui précède), on peut la reprendre dans la mosaïque.

Q32

```
127 import copy
128
129 def belleMosaique(source:image, vignettes:[image], p:int) -> image:
130     h, w = vignettes[0].shape
131     vign = copy.deepcopy(vignettes) # on conserve vignettes inchangé
132     envisagees= {}
133     img_mosaique = initMosaique(source, w, h, p)
134
135     for i in range(p):
136         for j in range(p):
137             pavé = img_mosaique[i*h:(i+1)*h, j*w:(j+1)*w]
138
139             ind = choixVignette(pavé, vignettes)
140             if ind not in envisagees: # vignette nouvellement utilisée
141                 envisagees[ind] = 1
142
143             if envisagees[ind]%10 == 0: # vignettes envisagée 10 jois (
    ou un multiple de 10)
144                 envisagees[ind] = 1
145                 vign.append(vignettes[ind])
146
147             while ind in envisagees and envisagees[ind]%10 != 0: # on
    enlève une vignette utilisée 1 fois et envisagée moins de 10 fois,
    jusqu'à trouver une bonne vignette
148                 envisagees[ind] += 1
149                 del vign[ind]
150                 ind = choixVignette(pavé, vign)
151
152             img_mosaique[i*h:(i+1)*h, j*w:(j+1)*w] = vignettes[ind]
153     return img_mosaique
```