Épreuve du jeudi 24 février 2022 — durée : 3h

CONCOURS BLANC D' INFORMATIQUE (PYTHON ET SQL)

——— TRAITEMENT D'IMAGE POUR LA ROBOTIQUE EMBARQUÉE ———

Corrigé

Partie 1. Image : inspiré d'extraits de PSI 2011

On dote un robot d'exploration d'une camera afin de tester la possibilité de reconnaitre des obstacles. Le flux vidéo est découpé en une suite d'images qu'un traitement annexe transforme en images en **nuances de gris** (elles ne sont donc plus en couleurs). Une image est ici la donnée d'une hauteur H, d'une largeur L, et d'une matrice M d'entiers de H lignes et L colonnes. L'image est divisée en éléments ou pixels définis par leurs numéros de ligne noté ℓ et de colonne noté c. Chaque entier de la matrice définit le ton de gris associé au pixel de coordonnées (ℓ, c). Ce ton varie entre zéro, qui rend l'absence de lumière et donc le noir, et une valeur maximale dite profondeur P qui rend le blanc. Les valeurs intermédiaires rendent diverses teintes de gris de plus en plus claires. On notera que le pixel de coordonnées $(0, 0)$ est conventionnellement situé en haut et à gauche de l'image.

On suppose que l'on peut créer une image par l'appel de fonction `allouer(H, L, P)` . On accédera aux composants d'une image i par les notations `i.H` (hauteur), `i.L` (largeur), `i.P` (profondeur de gris) et `i.M` (matrice). On accédera aux éléments de la matrice par la notation `i.M[`ℓ, c`]`, sachant que les indices commencent à zéro, un indice de ligne est donc un entier ℓ compris entre 0 et $H - 1$ au sens large, tandis qu'un indice de colonne est un entier c compris entre 0 et $L - 1$ au sens large.

1 On veut écrire une fonction `inverser(i)` qui renvoie l'image inverse (inversion de contraste, par exemple un pixel blanc devient noir, et réciproquement) de l'image i. Complétez la ligne 9 ci-dessous.

```
1  def inverser(i):
2  # Inverse les niveaux de gris d'une image
3          resultat = allouer(i.H,i.L,i.P)
4          t_original = i.M
5          t = resultat.M
6          profondeur = i.P
7          for j in range(i.H):
8                  for k in range(i.L):
9                    ... à compléter ...
10         return( resultat )
```

On utilise la profondeur `P` associé à l'image `i`. C'est la valeur maximum utilisée pour coder le gradient de gris entre 0 (le noir) et P (le blanc). Comme on veut inverser l'image, il faut transformer selon noir $\Longleftrightarrow$ blanc donc on va soustraire la valeur actuelle de gris du pixel à P pour inverser la teinte de gris.

```
1  def inverser(i):
2  # Inverse les niveaux de gris d'une image
3  resultat = allouer(i.H,i.L,i.P)  # on a une copie de l'image envoyée en paramètre i
4  t_original = i.M   # On accéde aux éléments de la matrice par la notation\mot{ i.M[$\ell, c$]}
5  t = resultat.M    # on fait de même avec notre copie : on peut travailler sur la matrice de pixels t
6  profondeur = i.P
7    for j in range(i.H): # pour chaque ligne
8      for k in range(i.L): # pour chaque colonne
9          t[j,k] = profondeur - t_original[j,k]  # on inverse le contraste
10   return( resultat )
```

2 Écrire une fonction `juxtaposer(i_1, i_2)` qui prend en arguments deux images i_1 et i_2 de même largeur et profondeur, et qui renvoie la nouvelle image obtenue en juxtaposant i_1 et i_2. L'image i_1 est à gauche dans la nouvelle image, i_2 est à droite.

Vous utiliserez la trame suivante, en complétant les lignes 4 et 8 à 11 :

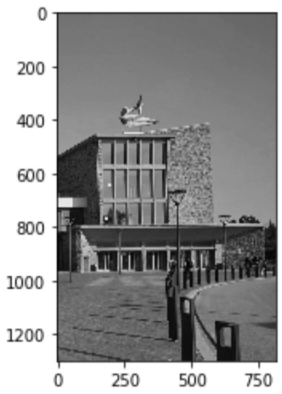
```
def justaposer(i1,i2):
  l1 = i1.L   ,  l2 = i2.L
  hauteur = i1.H
  resultat = allouer(   ... à compléter ...   ,i1.P)
  t1 = i1.M   ,  t2 = i2.M
  t = resultat.M
  for j in range(hauteur ):
    ...
    ...
    ...
    ...
  return( resultat )

```

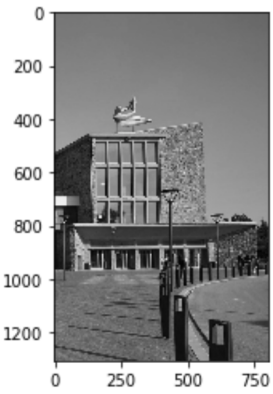
On utilise un tableau vide de la taille des deux images juxtaposées et on balaie ligne à ligne les deux images initiales pour coller deux lignes côte à côte

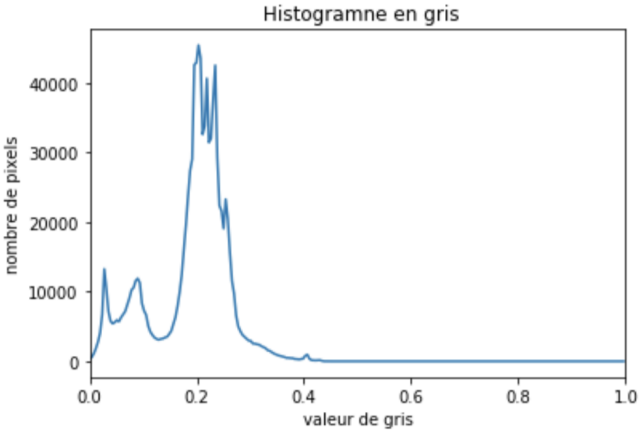
```
def justaposer(i1,i2):
  l1 = i1.L   ,  l2 = i2.L  # on stocke les largeurs des deux images, mais on sait que dans cet énoncé c'est identique
  hauteur = i1.H
  resultat = allouer( hauteur,l1+l2,i1.P) # on crée un tableau image de largeur correspond aux images côte à côte
  t1 = i1.M   ,  t2 = i2.M # on accède aux  tableaux des deux images
  t = resultat.M  # on accède au tableau de l'image finale (juxtaposition)
    for j in range(hauteur ):  # on balaie les lignes des images
      for k in range(l1):  # on balaye les colonnes de l'image de gauche et on la stocke dans le tableau final
        t[j,k] = t1[j,k]
      for k in range(l2):   # on balaye l'image de droite et on la stocke dans le tableau final
        t[j,k+l1] = t2[j,k]
  return( resultat ) # on renvoie le tableau image final
```

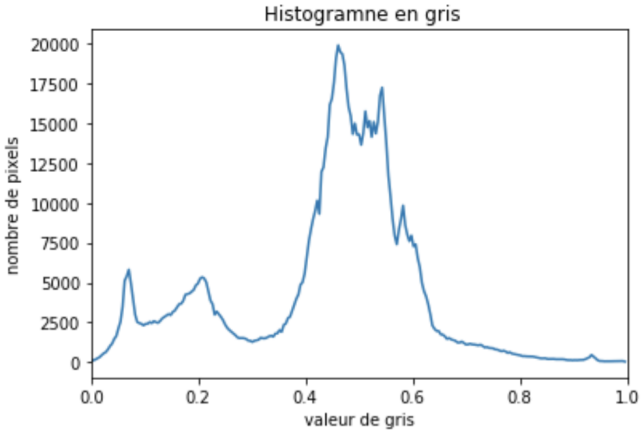
Au lieu de cette méthode d'école, on peut aussi utiliser les listes pour concaténer directement `ligne_1(j)` et `ligne_2(j)` des deux images, plutôt que de balayer chaque image pixel par pixel (avec `k`)

3 En traçant l'histogramme d'une image contrastée ou assombrie, on voit une diffférence dans le tracé, comme ci-dessous.



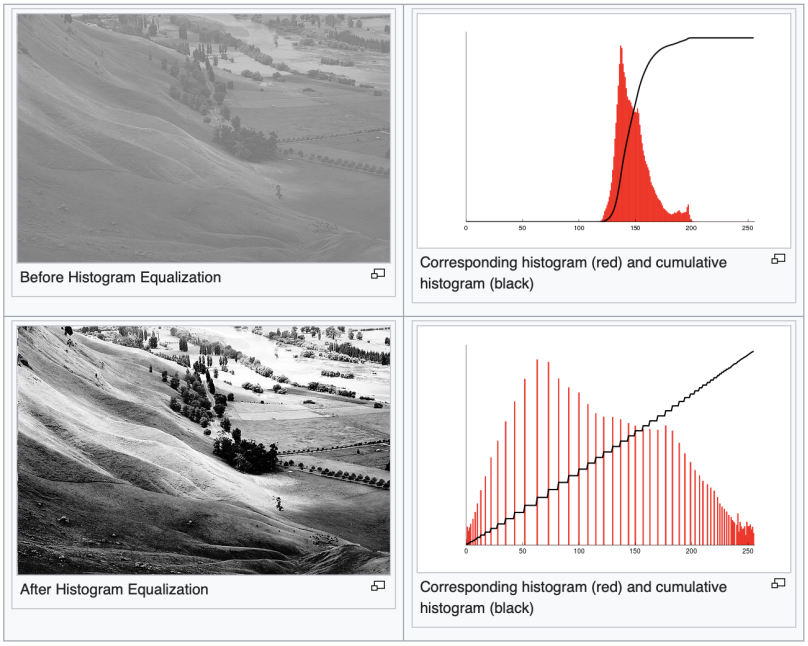






En abscisse, on a la valeur de gris notée entre 0 et P la profondeur, par exemple 256 pour 256 nuances de gris. En ordonnées, on a le nombre de pixels ayant cette valeur dans toute l'image. Quelle fonction mathématique semble cachée dans la différence de tracé des deux histogrammes ?
Comme le point à gauche `nombre-de-pixel(valeur-gris=0)` a la même valeur 0, la dilatation de la courbe semble correspondre à une homothétie de centre `valeur-gris=0`

4 On cherche maintenant à effectuer le traitement allant d'une image peu contrastée à une image contrastée. Le cas ci-dessous correspond à une image sur-exposée, donc trop claire, avec peu de contraste dans les teintes sombres.



Quel est l'algorithme à appliquer pour passer d'un histogramme non contrasté à un histogramme contrasté réparti sur toutes les valeurs de l'axe horizontal ?
Supposons que l'intervalle de valeurs de gris est de $min = 0$ à $max = 256 = P$ par exemple. Il vaut travailler directement avec l'histogramme : soit `histogramme(i)` celui de l'image passée en argument. Pour cette fonction, on peut envisager ceci :

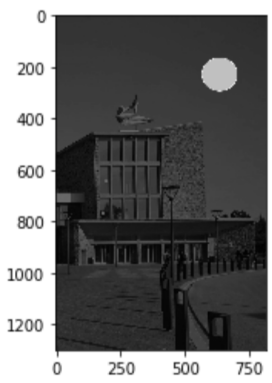
```
def histogramme(img):
  tableau = [0 for k in range(img.P+1)]  # tableau vide de la même largeur que le nombre de niveaux de gris
  for i in range(img.H):
    for j in range(img.L):  # pour tous les points de l'image img,
      tableau[img.M[i, j]] += 1   # on ajoute 1 au niveau de gris img.M[i, j] donné par le pixel de la boucle
  return tableau
```

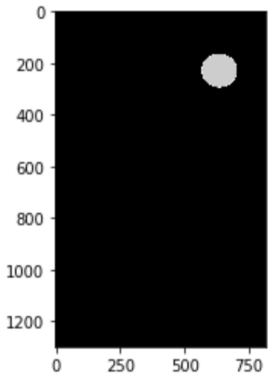
L'algorithme permettant de contraster l'image `img` en `image_2` fonctionne en utilisant cette dilatation :

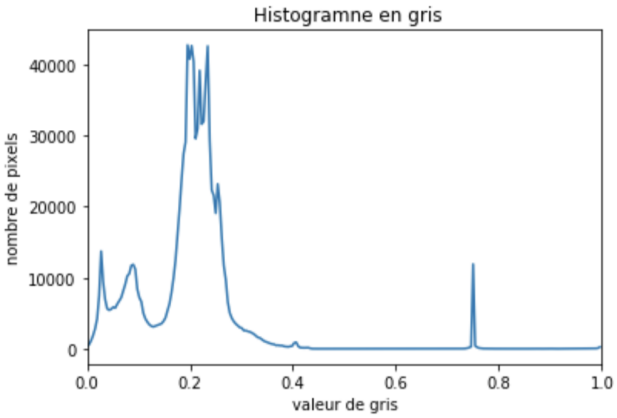
$$img\_2(i,j) = P \times \frac{img(i,j) - \text{ ValMin}}{\text{ValMax} - \text{ValMin}}$$

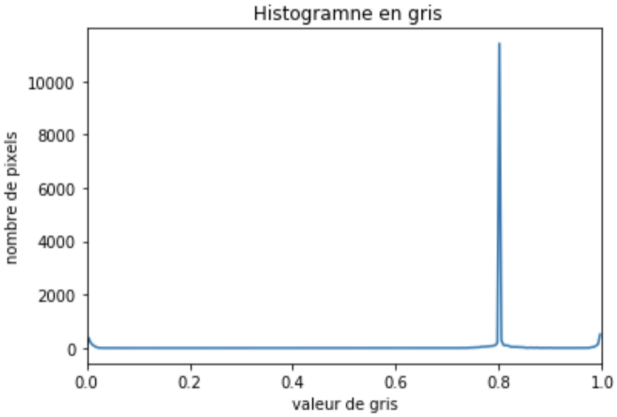
Mais pour cela, il faut fournir ou calculer les valeurs `ValMin` et `ValMax` qui correspondent aux frontières de l'histogramme de l'image non contrastée `img`

5 Un objet clair remarquable dont on veut suivre le mouvement passe dans le champ de la prise de vue. On veut isoler cet objet en le plaçant sur un fond noir.
Où est situé dans l'histogramme de gauche cet objet clair et uniforme ? Comment serait l'histogramme si l'objet n'était pas uniforme ?
L'objet clair est responsable du pic isolé à droite sur l'histogramme de gauche
L'objet clair n'était pas uniforme, on aurait une variation des teintes de gris de cet objet et on aurait donc un pic isolé à droite sur l'histogramme de gauche mais ÉLARGI









6 Montrer que la fonction à réaliser ressemble à un filtre passe-haut dont vous préciserez la coupure.
Pour passer de l'histogramme de gauche à celui de droite, on doit annuler les valeurs de `histogramme()` pour les valeurs de gris inférieures à celles du pic de l'objet clair par exemple `gris-pic`.On peut modéliser la fonction de transfert par 0 pour `gris` < `gris-pic` et 1 sinon.

7 On suppose que l'on récupère deux tableaux `pixels` et `gris` par l'appel de la fonction de bibliothèque `numpy` :
`pixels, gris = np.histogram(image, bins=256, range=(0, 1))`
Comment modifier le tableau `pixels` pour isoler l'objet clair en remplaçant le fond par du noir ?

```
def isole_objet(img, gris_pic):
  image = [img.M(i,j) for i in range(img.L+1) for j in range((mg.H+1)]  # copie de l'image " en liste par compréhension"
  for i in range(img.H):
    for j in range(img.L):  # pour tous les points de l'image img,
      test = image.M[i, j] < gris_pic  # le pixel est plus sombre ? vaut TRUE ou FALSE
      modif = int(test) # converti en 0 ou 1
      image.M[i, j]= modif* img.M[i,j]  # on met les pixels à 0 s'ils sont plus sombres que l'objet clair
  return image
```

8 Pourquoi doit-on modifier `pixels[0]` avant l'appel du tracé ? Quelle valeur doit-on donner ?
On a peut-être trop de pixels noirs donc le nombre est trop grand et va tasser l'histogramme en ordonnée, on fait donc pixels[0]=0 # suppression du noir, attention la fonction `histogramme` est remplacée ici par `pixels` qui est l'un des rendus de la fonction `numpy` qui est `histogram`

Partie 2. Tri des listes de valeurs de gris

On veut maintenant savoir si une valeur de gris est utilisée pour un ou des pixels dans une image fournie.
On veut donc trouver si un élément recherché existe bien dans une liste. S'il existe, on fournit l'indice de la première occurrence trouvée. Sinon, il renvoie une réponse vide.

9 Comment passer d'un tableau de valeurs de gris comme précédemment à une liste de valeurs de gris dans l'ordre des lignes et colonnes ?
On lit avec deux `for` le tableau-image et on le range au fur et à mesure dans la liste par un `append` en partant d'une liste vide `liste[ ]`

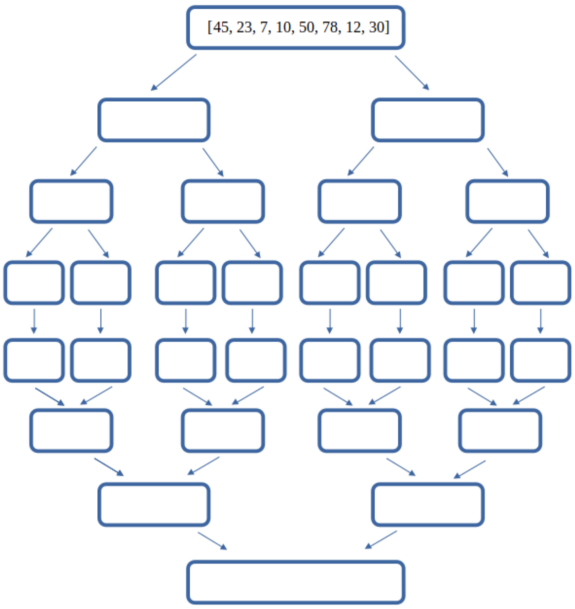
10 Dans une liste **triée**, on regarde la valeur centrale dans les indices encore disponibles. S'il ne s'agit pas de la bonne valeur, on enlève de l'intervalle de recherche les indices qui ne peuvent pas contenir la valeur recherchée. L'algorithme se décrit ainsi :

```
g = 0
d = longueur - 1
TANT QUE g <= d
c = (g + d) // 2
SI liste[c] < x
on ne garde que ce qui est à droite de l'index centre
g = (c + 1)
SINON SI liste[c] > x
on ne garde que ce qui est à gauche de l'index centre
d = (c - 1)
SINON
Si on arrive ici, c'est que x est à l'index centre
Renvoyer c
Fin du Si
Fin Tant que
Si on arrive ici, c'est que l'intervalle de recherche est vide
Renvoyer VIDE
```

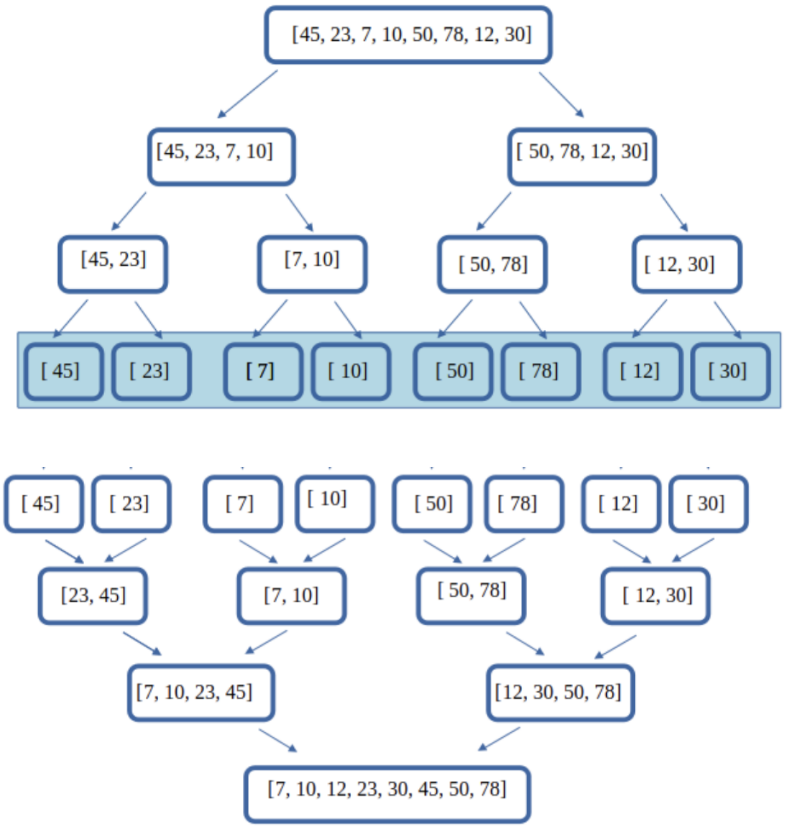
Commentez le terme de "recherce dichotomique" utilisé pour décrire cette méthode.
Soient une liste triée par ordre croissant d'éléments et un élément x, on cherche à déterminer si x se trouve dans `liste`. La démarche est connue, et vous l'avez déjà étudié en première année : il s'agit de comparer x à `liste[c]` avec $c = round(n/2)$ puis :
- si $x = liste[c]$, la recherche est terminée ;
- si $x < liste[c]$ la recherche se poursuit dans $t[0 \dots c-1]$;
- si $x > liste[c]$ la recherche se poursuit dans $t[c+1 \dots d]$.

On découpe donc en deux (dichotomie) à chaque test afin de réduire l'intervalle de recherche.

11 Nous allons donc utiliser le tri fusion qui consiste à :
— Phase DIVISER : on divise en deux notre tableau, récursivement.
— Phase REGNER : on règne sur le problème du tri lorsqu'on obtient... un tableau de une case. Elle est triée !
— Phase COMBINER : on fusionne alors facilement nos deux tableaux triés : le plus petit élément de chaque tableau est au début du tableau. Trier les éléments des deux tableaux revient donc à juste à chaque étape à lire deux valeurs et faire leur comparaison.



Remplir les cases de ce tri_fusion
On a une phase de séparations puis une phase de fusion :



12 Compléter le code de `tri_fusion()` suivant aux lignes 5 et 6 :

```
1    def tri_fusion(t):
2      n = len(t)
3      if n < 2:
4        return t
5      a =   ......
6      b =   ......
7      return fusion(a, b)
8
```

On utilisera le code de la fonction intermédiaire `fusion()`

```
1  def fusion(a, b):
2    p, q = len(a), len(b)
3    c = [None] * (p + q)
4    i=j=0
5    for k in range(p+q):
6      if j >= q:
7        c[k:] = a[i:]
8        break
9      elif i >= p:
10       c[k:] = b[j:]
11       break
12     elif a[i] < b[j]:
13       c[k] = a[i]
14       i += 1
15     else:
16       c[k] = b[j]
17       j += 1
18   return c
19
```

On a alors :

```
1  def tri_fusion(t):
```

```
    n = len(t)
    if n < 2:  # plus qu'un seul élément, on arrête le découpage
      return t
    a = tri_fusion(t[:n//2])  # on trie la gauche
    b = tri_fusion(t[n//2:])  # on trie la droite
    return fusion(a, b)
```

13 Quelle est la complexité de la fonction `fusion()` ?
Quel est le coût temporel ? La fonction `fusion` est de coût linéaire vis-à-vis de la somme des longueurs des deux tableaux passés en paramètre. Si $C(n)$ désigne le coût du tri d'un tableau de longueur n, on dispose de la relation :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

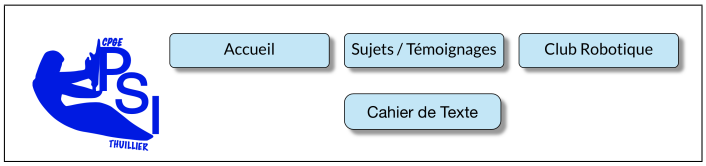
Ce type de relation de récurrence est typique des méthodes dites "diviser pour régner" ;on prouve que cette relation implique que $C(n) = \Theta(n \log n)$. Il n'est pas possible de faire mieux dans le cadre des algorithmes de tri par comparaison.

Partie 3. Traitement des images envoyées par la camera

Nous allons reprendre le flux d'images extraites de la vidéo, mais cette fois nous utiliserons les images en couleurs. Nous aurons besoin des deux bibliothèques suivantes : `numpy` et `matplotlib.pyplot` :

```
1  import numpy as np
2  import matplotlib.pyplot as plt
```

La seconde contient deux fonctions qui vont nous être utiles :
- la fonction `plt.imread` prend en argument une chaîne de caractère décrivant le chemin d'accès à un fichier image au format PNG par exemple et retourne un tableau NUMPY. Ce tableau a même dimension que l'image, et chacune de ses cases contient un triplet donnant une liste (triplet) des composantes RGB du pixel correspondant ;
- la fonction `plt.imshow` prend en argument un tableau NUMPY et affiche l'image associée à cette matrice, éventuellement suivie de l'instruction `plt.show ( )` si le mode interactif n'est pas activé.

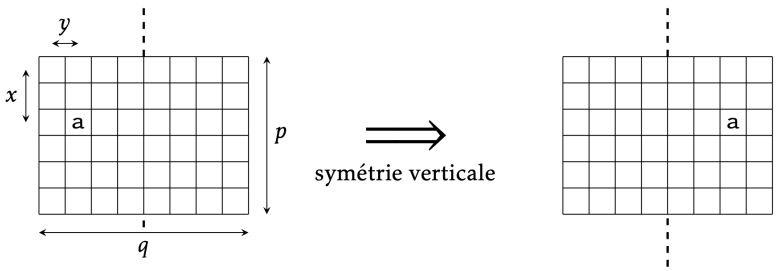


On affiche l'image ci-dessus grâce au code suivant :

```
1  site = plt.imread('sitepsi.png')
2  plt.imshow(site)
```

Les dimensions $p \times q$ de l'image initiale peuvent être calculées par les instructions `p=img.shape[0]` et `q=img.shape[1]`. Ici, pour l'image ci-dessus, on a 578×2824

14 Dans le cas d'une symétrie d'axe vertical, l'image transformée a les mêmes dimensions que l'image initiale. Pour créer la matrice-image vierge, on utilisera l'instruction `np.zeros_idem(img)` qui prend en argument une matrice `img` et qui renvoie une matrice vierge de mêmes dimensions.



Pour des besoins de symétrie d'image lorsque l'on donne l'ordre au robot d'inverser son sens de déplacement, étant donné un pixel `a` de coordonnées (x, y) dans l'image initiale, comme représenté sur le schéma précédent, exprimer ses coordonnées (x', y') dans l'image résultat d'une symétrie d'axe vertical passant par le centre de l'image.
Pour la symétrie du schéma, on a $x' = x$ et $y' = q - 1 - y$

15 On va construire une fonction `symetrie(img)` qui prend en argument une matrice-image `img` et qui renvoie une nouvelle matrice-image résultat de la symétrie d'axe vertical. Les dimensions $p \times q$ de l'image initiale peuvent être calculées par les instructions `p=img.shape[0]` et `q=img.shape[1]`. Complétez le code suivant aux lignes 4 et 5 en utilisant le fait que `img[x, y]` correspond à une sous-liste `[r,v,b]` des 3 couleurs RVB de ce pixel à la ligne x et la colonne y de ce tableau.

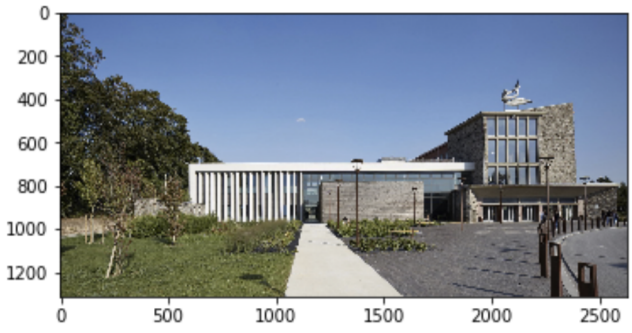
```
0  def symetrie(img):
1    p, q = img.shape[0], img.shape[1]
2    img2 = np.zeros_idem(img)
3    for x in range(p):
4
5
6    return img2
```

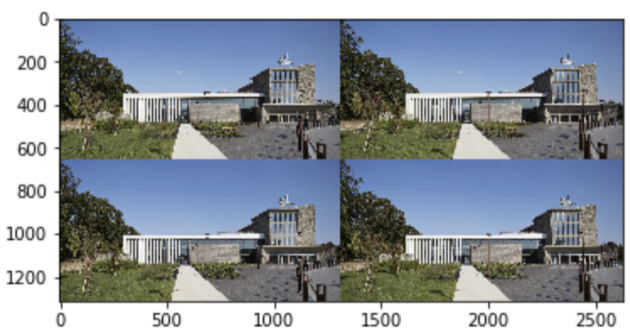
On a ici un avantage, cest que `img[x, y]` contient directement la sous-liste de couleur RVB `[r,v,b]` et on a donc à simplement copier `img[x, y]` à la bonne position de la nouvelle image.

```
0 def symetrie(img):
1   p, q = img.shape[0], img.shape[1] # largeur et hauteur de l'image img
2   img2 = np.zeros_idem(img)  # création de l'image vide initiale
3   for x in range(p):
4     for y in range(q): # pour toute ligne et toute colonne
5       img2[x, q-1-y] = img[x, y]  # on utilise les formules de symétrie
6   return img2
```

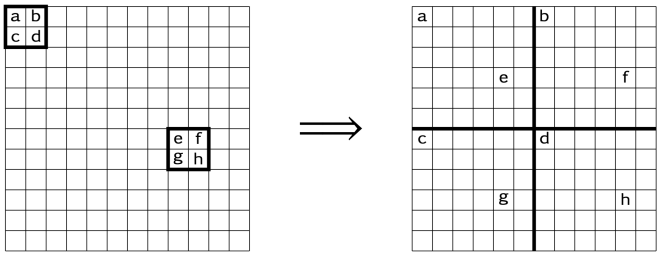
16 Comment coder la fonction `np.zeros_idem(img)` ? Pourquoi la ligne 1 du code précédent semble redondante ?
On définit deux boucles sur les lignes et les colonnes et on met à zéro à la main : `img2[x, y] =0`

17 Comme le contrôleur humain du robot a un retour vidéo de la camera embarquée, il est intéressant de visualiser sur l'écran les différents résultats de traitements d'image : on veut placer les différentes images modifiées par filtrage et traitement (par exemple 3) en plus de l'image initiale sur les 4 parties de l'écran, divisée en zones égales. C'est ensuite sur les quarts d'écran que seront effectués les filtrages et traitements, comme par exemple la reconnaissance d'objets. Montrons le principe avec pour le moment la copie de quatre fois la même image en couleurs de dimension 1312×2632.





On a une **contrainte** : comme vous le voyez sur le quadrillage des images précédentes, l'image finale a la même taille que l'image initiale. Il ne s'agit donc pas de recopier (fonction `crop()` en Python) l'image initiale quatre fois, sinon on obtient une image quatre fois plus grande, ce qui encombre la mémoire. Une méthode serait donc de faire la moyenne d'un paquet de 4 pixels sur le dessin de gauche (voir le schéma `abcd` en dessous) et de le placer aux positions des pixels notés a, b, c et d du dessin de droite. On ne choisit pas cette méthode.
Pour que cette transformation soit bijective, on a découpé l'image initiale en paquets carrés de quatre pixels (2×2) puis pour chaque paquet carré de quatre pixels, on utilise celui en haut à gauche pour l'image réduite en haut à gauche, celui en haut à droite pour l'image réduite en haut à droite, etc.
On notera que pour que cette transformation soit facilement définie il est nécessaire que les dimensions horizontale et verticale de l'image soient paires et qu'une image carrée $2n \times 2n$ serait idéale à traiter.



La méthode utilise une transformation qui utilise les formules :

$$(x',y') = \begin{cases} (x/2, y/2) & \text{si } x \text{ et } y \text{ sont pairs} \\ (x/2, y/2+q/2) & \text{si } x \text{ est pair et } y \text{ impair} \\ (x/2+p/2, y/2) & \text{si } x \text{ est impair et } y \text{ pair} \\ \textbf{... à compléter ...} & \text{si } x \text{ et } y \text{ sont impairs} \end{cases}$$

On a :

$$(x',y') = \begin{cases} (x/2, y/2) & \text{si } x \text{ et } y \text{ sont pairs} \\ (x/2, y/2+q/2) & \text{si } x \text{ est pair et } y \text{ impair} \\ (x/2+p/2, y/2) & \text{si } x \text{ est impair et } y \text{ pair} \\ \textbf{(x / 2+p / 2, y / 2+q / 2)} & \text{si } x \text{ et } y \text{ sont impairs donc } (\frac{x}{2}+\frac{p}{2}, \frac{y}{2}+\frac{q}{2}) \end{cases}$$

On rappelle que la division entière est notée `//` et que le reste est noté `%`. Ainsi, `5 // 2 = 2` et `5 % 2 = 1`. En quoi l'écriture de la fonction suivante `distribution` plus concise, à deux tests au lieu de 4, permet de traduire la transformation (x',y') ci-dessus ? Complétez la ligne 4.

```
0  def distribution(k, d):
1    if k % 2 = = 0:
2      return k // 2
3    else:
4      return  ... à compléter ...
```

La fonction `distribution` va permettre de traiter les lignes 1 et 2, ou alors 3 et 4 de la description de `x'` ou `y'`.

```
0  def distribution(k, d):  # on envoie les données  ()x ou y , p ou q)
1    if k % 2 = = 0:  # si k est pair
2    return k // 2      #
3    else:                 # si k est impair
4    return k // 2 + d // 2
```

18 Dans le code précédent, on utilise la division entière "//" au lieu de la division décimale "/". Pourquoi ?
Les pixels ont des positions entières et on veut orienter selon que `x` et `y` sont pairs ou impairs. Mais le résultat de la ré-affectation des pixels dans l'image finale doit être entier.

19 Si, comme dans certains autres langages, le Python ne possédait pas cette instruction "//", comment pourrait-on la réaliser avec la fonction `int()`. Rappelons que `int(3.6)` donne `3`. Comment pourrait-on obtenir l'opérateur `%` en écrivant une fonction `reste(a,b)` basée sur certaines des 4 seules opérations + - * / et également la fonction `int()` ?
Comme la fonction `int()` réalise la troncature du réel sans arrondir à la valeur supérieure, le résultat de la division entière peut être simulé par `int( a/b )`
Pour la fonction `rest(a,b)`, il faut utiliser la fonction `int()` : on sait que $a = bq + r$ et on a déjà $b = int(a/b)$ donc le reste est $r = a - b \times q$ donc `reste(a,b)` = a - b × `int(a/b)`

20 Si l'on ne veut pas utiliser `int()` dans `reste(a,b)`, permettant de déterminer le reste de la division euclidienne d'un entier naturel a par un entier naturel non nul b, on peut utiliser l'algorithme simpliste :
entrées : a et b entiers naturels, avec $b \neq 0$
initialiser une variable q à 0
initialiser une variable r à a
tant que r $\geqslant b$:
  remplacer q par q+1
  remplacer r par r $- b$
sortie : la valeur finale de r
Traduire cet algorithme en fonction `reste2(a,b)` dans le langage Python.
Traduction en Python :

```
0 def reste2(A,B):
1   q , r = 0, A  # quotient et reste initialisés à 0 et A
2   while( r >= B): # tant que le reste est divisible par B
3     q +=1  # le nombre de fois B augmente
4     r = r - B  # on diminue le reste
5     # optionnel pour voir les étapes  print(q,r)
6   return r  # on renvoie le  reste r mais on peut aussi envoyer r et q avec return q,r
```

On obtient alors avec le `return q,r` :
> reste2 (20, 3)
>1 17

>2 14
>3 11
>4 8
>5 5
>6 2

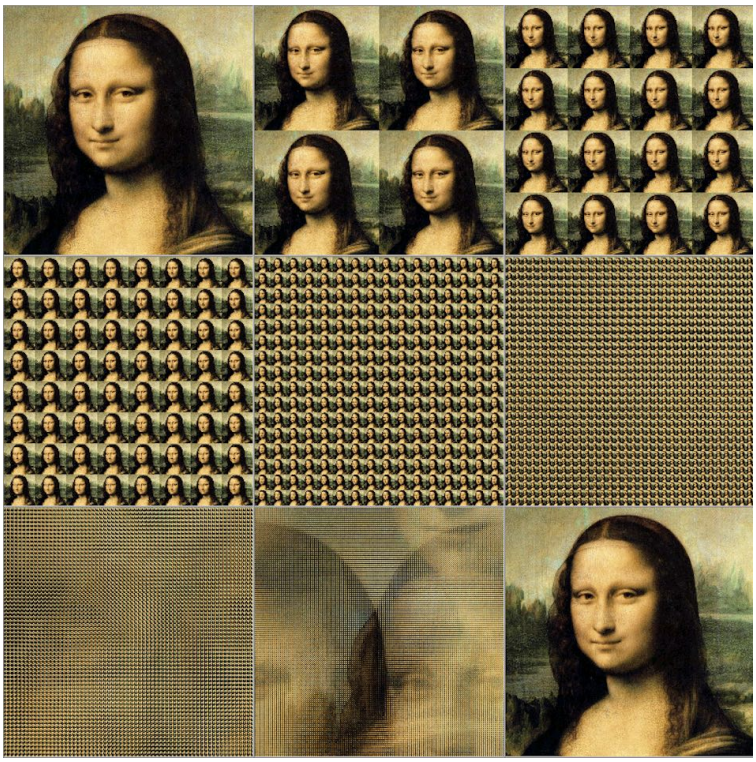
21 On va maintenant utiliser la fonction `distribution` pour obtenir un tableau correspondant à l'image composée par le critère (x', y'). Complétez la ligne 5 de la fonction `quatre_vues` ci-dessous.

```
0  def quatre_vues(img):
1    p, q = img.shape[0], img.shape[1]
2    img2 = np.zeros_idem(img)
3    for x in range(p):
4      for y in range(q):
5        img2[  ...  ,    ...   ] = img[x, y]
6    return img2
7
```

La fonction `distribution(k,d)` porte sur : x ou y en premier argument `k` , et sur p ou q comme 2e argument `d`

```
0 def quatre_vues(img):
1   p, q = img.shape[0], img.shape[1]  # largueur et hauteur de l'image initiale
2   img2 = np.zeros_idem(img)
3   for x in range(p):  # pour chaque point de l'image initiale
4     for y in range(q):
5       img2[distribution(x, p), distribution(y, q)] = img[x, y]  # on décale les pixels dans l'une
      des quatre parties
6   return img2
```

22 Il existe une particularité non abordée pour cette méthode que nous venons de développer : des appels successifs de la fonction `quatre_vues` sur l'image précédemment donnée par cette même fonction donnent une évolution de l'image perçue comme sur le cas ci-dessous, avec comme image initiale une partie carrée $2n \times 2n$ du tableau de la Joconde.



Au bout d'un certain nombre d'appels itératifs de `quatre_vues`, l'image de départ ré-apparait clairement. Ce nombre d'appels nécessaires dépend de quel(s) paramètre(s) ?
Cette image est de taille $L \times H$. Le plus petit entier n pour lequel $L-1$ divise 2^n-1 est n_1, tandis que le plus petit entier n pour lequel $H-1$ divise 2^n-1 est n_2. Donc l'image de fin ci-dessus n'est pas forcément égale à l'image initiale. Des pixels peuvent être "éparpillés"

23 Peut-on envisager d'utiliser ces observations pour cacher un texte dans l'image de la Joconde, en supposant qu'il existe une fonction `ajout(texte, coordonnees [x,y], image)` ? Comment procéder pour utiliser `ajout()` ?
Données : `coordonnees [x,y]` correspond à la position dans l'image du texte à partir du pixel `[x,y]` qui est la partie en haut à gauche de l'image équivalente au texte et `image` est l'image sur laquelle porte cet ajout de texte. Le retour de `ajout()` est un tableau image.
La fonction `ajout` doit convertir les lettres du mot en pixels, et les remplacer dans l'image initiale. Ensuite, des appels itératifs de `quatre_vues` permettent de "perdre" les pixels des lettres, puis de les rassembler approximativement par de nouveaux appels récursifs.

Partie 4. Codage binaire à améliorer en vue de transmissions sans erreurs

Le codage binaire (avec 0 et 1) doit parfois être consolidé. Avec deux niveaux de tension, l'échange des données d'une image sous la forme d'une suite de nombres binaires peut comporter des erreurs de transmission. Des algorithmes de détection ou correction d'erreur existent.
Il a été testé une **autre méthode** : le codage non pas en base 2 mais EN BASE 3.
En base 3 classique, on écrit les nombres à l'aide de trois chiffres : 0,1 et 2 : par exemple, $23 = \overline{212}^3 = 2\times3^2+1\times3+2$ ou encore $46 = \overline{1201}^3 = 1\times3^3+2\times3^2+0\times3+1$.
En électronique, les tensions nécessaires pour le 0 et le 1 informatique existent déjà. Afin de ne pas modifier trop les sources de tensions, on préfère cette base 3 :
en base 3 modifiée, les "chiffres" utilisés sont $0, 1$ et -1, et on notera m la base :

$$23 = \overline{10(-1)(-1)}^m = 1\times3^3+0\times3^2-1\times3-1$$
$$46 = \overline{1(-1)(-1)01}^m = 1\times3^4-1\times3^3-1\times3^2+0\times3+1$$

Dans toute la suite, on s'intéresse à l'écriture en base 3 modifiée des entiers naturels non nuls. On note $C = \{-1, 0, 1\}$ l'ensemble des chiffres utilisés dans l'écriture en base 3 modifiée.
L'écriture en base 3 modifiée d'un entier naturel non nul sera notée comme ci-dessus, par la suite des chiffres surmontée d'un trait avec un exposant e. Si $a_0, a_1, \ldots, a_{p-1}$ sont dans C, on a donc :

$$\overline{a_0a_1\ldots a_{p_1}}^m = \sum_{k=0}^{p-1} a_k 3^{p-1-k}$$

L'écriture en base 3 modifiée d'un entier naturel non nul, lue de gauche à droite, commence toujours par le chiffre 1.
24 Donner la valeur de $\overline{1(-0)0(-1)}^m$, de $\overline{1111}^m$ et de $\overline{1(-1)(-1)(-1)(-1)}^m$
Faute de frappe : lire $\overline{1(-1)0(-1)}^m$ mais j'ai compté bon le $\overline{1(0)0(-1)}^m$
$1(-1)0(-1) = 1*3^3+(-1)*3^2+0*3-1 = 27-9-1 = 17$ et on aurait $1(0)0(-1) = 1*3^3+0*3^2+0*3-1 = 27-1 = 26$
$1111 = 1*3^3+1*3^2+1*3+1 = 27+9+3+1 = 40$
$1(-1)(-1)(-1)(-1) = 3^4-3^3-3^2-3-1 = 81-27-9-3-1 = 41$
25 Écrire en base 3 modifiée les entiers de 1 à 9.
On aboutit à :

base 10	base m	Explications
1	1	$=1$
2	$1(-1)$	$=3-1$
3	10	$=3$
4	11	$=3+1$
5	$1(-1)(-1)$	$=9-3-1$
6	$1(-1)0$	$=9-3$
7	$1(-1)1$	$=9-3+1$
8	$10(-1)$	$=9-1$
9	100	$=9$

26 Soit k un entier naturel. Déterminer la valeur de $A_k = \overline{1\ \underbrace{11\ldots1}_{k \text{ chiffres}}}^m$ et de $B_k = \overline{1(-1)(-1)\ldots(-1)}^m_{k+1 \text{ chiffres}}$ (On aura bien noté que ces deux écritures possèdent $k+1$ chiffres au total.)
$A_k = 1\ 1...1 = 3^k+3^{k-1}+\ldots+3+1 = \left(3^{k+1}-1\right)/2$
$B_k = 1(-1)\ldots(-1) = 3^k-3^{k-1}-\ldots-3-1 = 3^k-\left(3^k-1\right)/2 = \left(3^k+1\right)/2$

27 On représente l'écriture en base 3 modifiée $\overline{1(-1)(-1)(-1)(-1)}^m$ par la liste Python $[a_{p-1}a_{p-2},\dots,a_0]$ (attention, les chiffres sont écrits dans la liste en lisant de droite à gauche l'écriture en base 3 modifiée, le dernier d'entre eux est donc toujours le chiffre 1). **On veut écrire** à l'aide du code ci-dessous une fonction `valeur`(L) qui prend en argument une liste de chiffres et renvoie l'entier naturel non nul dont c'est une écriture en base 3 modifiée.
Par exemple `valeur`$([1,0,-1,-1,1])$ renvoie l'entier 46.
Vous allez vous aider de la fonction Python `enumerate` qui fonctionne de la manière suivante : le code suivant

```
0 for compteur, valeur in enumerate([1,0,-1]):
1   print(compteur, valeur)
```

donne une sortie
0 1
1 0
2 -1
Il faut faire ici attention au fait que la liste doit comporter la décomposition dans le sens inverse à celui que l'on écrit depuis le début : on voit que le compteur est croissant de 0 à 2 alors que la puissance de 3 dans la définition varie en décroissant

$$\overline{a_0a_1\dots a_{p_1}}^m=\sum_{k=0}^{p-1}a_k3^{p-1-k}$$

Ainsi le nombre 46 cité correspond à la liste [1,0,-1,-1,1], mais c'est le codage $1(-1)(-1)01=1\times3^4+(-1)\times3^3+(-1)\times3^2+0\times3^1+1\times3^0=81-27-9+0+1=46$
On va donc implémenter la fonction demandée à partir de la trame suivante :

```
0 def valeur(L):
1 return sum(a * 3 ** index for           in enumerate(L))
```

Expliquer ce que doit contenir la boucle `for`
En supposant que les listes ont bien été initialement inversées, on a donc pour par exemple `L = [1,0,-1,-1,1]` :

```
0 def valeur(L):
1   return sum(a * 3 ** index for index, a in enumerate(L))
```

28 On veut montrer que tout entier naturel non nul n admet une écriture en base 3 modifiée. Soit $n\in\mathbb{N}^*$. On note q et r le quotient et le reste dans la division euclidienne de n par 3 : $n=3q+r$ et $r\in\{0,1,2\}$. On suppose que q est non nul et admet une écriture en base 3 modifiée $q=\overline{a_0\dots a_{p-1}}^m$. Déterminer une écriture en base 3 modifiée de n dans le cas où $r=0$ ou $r=1$.
On suppose qu'un entier n est tel que : $n=3q+r$. On pose q sous la forme $(a_0\dots a_{p-1})$. On a ainsi n qui s'écrit : $(a_0\dots a_{p-1}0)$ si $r=0$ ce qui se traduit concrètement par le fait que multiplier par 3 revient à ajouter un zéro à droite.

Dans le cas de $r=1$, on pose q sous la forme $(a_0\dots a_{p-1})$. On a donc $(a_0\dots a_{p-1}0)$ pour $r=0$ et ici on ajoute le reste $r=1=3^0$ au résultat donc la notation est $(a_0\dots a_{p-1}1)=\overline{a_0\dots a_{p-1}1}^m$
29 On suppose que q est non nul et que $q+1$ admet une écriture en base 3 modifiée $q+1=\overline{b_0\dots b_{p-1}}^m$. Déterminer une écriture en base 3 modifiée de n dans le cas où $r=2$.
On suppose que $q>0$. Donc $q+1$ s'écrit $(b_0\dots b_{p-1})$. Si $r=2, n=3(q+1)-1$ alors il s'écrit $(b_0\dots b_{p-1}(-1))$ car multiplier par 3 met un 0 à droite qui est ensuite remplacé par le -1 puisque c'est -1×3^0.
30 Montrer par récurrence que tout entier naturel non nul admet une écriture en base 3 modifiée.
- Initialisation : On a vu au début du sujet que les premiers entiers non nuls possèdent une écriture en base 3 équilibrée.
- ordre n supposé vrai et la multiplication par 3
- Hérédité : On vient de voir comment passer de l'écriture de n à celle de $3n$ ou $3n+1$ (ce qui prouve que ces entiers ont une écriture), et également comment on peut passer de l'écriture de $3n+3$ à celle de $3n+2$. Mais on peut passer de l'écriture de $n+1$ à celle de $3n+3$ ce qui prouve que $3n+3$ et $3n+2$ ont une écriture.

31 On souhaite écrire en Python une fonction `incremente(L)` qui prend en argument une liste L de chiffres représentant une écriture en base 3 modifiée d'un entier naturel non nul n et qui renvoie une liste représentant une écriture en base 3 modifiée de $n+1$.
On propose la fonction récursive `incrementeR(L)` suivante.
Attention, ici on veut une fonction récursive.
Multiplier par 3 transforme (-1)1111 en (-1)11110 et diviser par trois transforme (-1)11110 en (-1)1111 MAIS n'oublions pas que la liste L doit être écrite à l'envers de la notation donc `(-1)1111` est une liste `[1,1,1,1,-1]`

```
0  def incrementeR(L) :
1    if len (L)==0 :
2      return [1]
3    elif L[0]==0 :
4      return [1]+L[1:]
5    elif L[0]== -1 :
6      return [0]+L[1:]
7    else :
8    return [-1] + incrementeR(L[1:])
```

Expliquer les lignes 7 et 8.
Explication de la ligne 7 : on est dans le cas où $n = n' + 1$ ($L[0]$ vaut 1 quand on est au `else`)
Explication de la ligne 8 : $n = n' + 1$ est représenté par la liste L. Donc $[0] + L[1:]$ représente n' et donc $L[1:]$ seul est un décalage à gauche de n' et représente $n'/3$. L'appel récursif représente alors $n'/3 + 1$ et en redécalant à droite on a 3^* $(n'/3 + 1)$ soit $n' + 3$ et en ajoutant -1 on obtient $n' + 2$ soit $n + 1$.
Pourquoi a-t-il fallu écrire les lignes 1 et 2 ?
Les lignes 1 et 2 sont là pour le cas terminal : à chaque appel récursif, la longueur de L diminue strictement de 1 . En partant d'une liste non vide on finira par tomber sur le cas len (L) == 0.

32 On veut réaliser une version itérative de la fonction `incremente(L)` . Expliquer ce que l'on doit coder en lignes 13 et 14 dans le code ci-dessous.

```
0  def incremente(L):
1    p = len(L)
2    M = []
3    k = 0
4    while k < p and L[k] = = 1:
5      M.append(-1)
6      k += 1
7    if k = = p:
8        M.append(-1)
9    elif L[k] = = 0:
10       M.append(1)
11       M = M + L[k+1:]
12   elif L[k] = = -1:
13
14
15   return M
```

```
0  def incremente(L):
1    p = len(L)
2    M = []
3    k = 0
4    while k < p and L[k] = = 1:  # cas d'un 1
5      M.append(-1)
6      k += 1
7    if k = = p:
8        M.append(-1)
9    elif L[k] = = 0:   # cas d'un 0
10       M.append(1)
11       M = M + L[k+1:]
12   elif L[k] = = -1:  # cas d'un -1
13       M.append(0)
14       M = M + L[k+1:]
15   return M
```

33 Pour un résultat de type `[1,0,-1,1,0,0,-1,1,1]`, correspondant au nombre N, pouvez vous prédire le résultat de `valeur(-N)` ?
`[1,0,-1,1,0,0,-1,1,1]` correspond à 8038 et `[-1,0,1,-1,0,0,1,-1,-1]` correspond à -8038 = on change les signes de tous les termes de la décomposition

34 On souhaite montrer que l'écriture en base 3 modifiée d'un entier naturel n non nul est unique. Soit $a_0, \dots, a_{p-1}$ des chiffres de C (avec $a_0 = 1$ et $p \geqslant 1$). Quel est le reste dans la division euclidienne par 3 du nombre $\overline{a_0 \dots a_{p-1}}^m$?
Si $a_0, \dots, a_{p-1}$ est la décomposition de C, alors C est la somme des $a_k 3^k$ qui sont tous divisibles par 3 sauf a_{p-1}.C et a_{p-1} ont donc le même reste dans la division euclidienne par 3 . Le reste de C est donc égal à a_{p-1} si celui-ci est positif et à a_{p-} $1 + 3 = 2$ sinon.

35 En déduire que l'écriture en base 3 modifiée d'un entier naturel n non nul est unique.
Si $a_0, \dots, a_{p-1}$ et $b_0, \dots, b_{p-1}$ désignent les écritures de C en base 3 équilibrée, on veut prouver que $q_k = b_k$ pour toute valeur de k. On peut le faire par récurrence décroissante sur k : D′ après la question précédente, on a nécessairement $a_{p-1} = b_{p-1}$, et on continue la récurrence après divisions successives par 3.

36 On veut écrire une fonction `base3(n)` qui prend en argument un entier naturel n non nul et renvoie la liste L de chiffres qui correspond à l'écriture en base 3 modifiée de n.
Par exemple `base3(46)` renvoie $[1, 0, -1, -1, 1]$.
Vous compléterez le code ci-dessous en lignes 6 et 7.

```
0  def base3(n):
1    chiffres = [0,1,-1]
2    ajout = [0, 0, 1]
3    L = []
4    while n > 0:
5      reste = n % 3
6      L.append(       )
7      n =
8    return L
```

```
0 def base3(n):
1   chiffres = [0,1,-1]
2   ajout = [0, 0, 1]
3   L = []
4   while n > 0:
5     reste = n % 3
6     L.append(chiffres[reste])
7     n = n // 3 + ajout[reste]
8   return L
```

A quoi sert la ligne 2 avec `ajout` ?
Celà permet de rendre plus lisible et court le choix du reste dans les 3 cas possibles

— FIN —