TP 1

Un premier TP de révision, l'objectif est de reprendre en main Python, les Bibliothèques, les programmations de base (récursives et autres)

Vous trouverez les documents, TP, cours corrections et annales sur le site :
https ://padlet.com/jbyvernault/ipt-decours-mp-psi-ggkcw07mqi1rv3xf

1 Echauffement

On importera la bibliothèque `numpy` pour utiliser les matrices : Pour créer une matrice :

```
A=np.array([[...,....],[....]])
```

On peut aussi transformer une liste en matrice

```
A=np.array([[ i +j for i in range(1,n)]for j in range(1;n)])
```

Parmi les opérations utiles :

```
A.shape() # renvoie la dimension de la matrice
n,p=np.shape(A)
np.dot(A,B)
A.dot(B) # pour multiplier deux matrices
np.transpose(A) # la transposee
A.T # transpose aussi
np.trace(A) # la trace
```

Les matrice usuelle pour en construire d'autres :

```
np.zeros((n,p)) # une matrice de 0 de taille (n,p)
np.ones((n,p) # une matrice de 1 de taille (n,p)
np.eye(n) # une matrice identite de taille (n,n)
np.diag([d_1,...d_n]) # renvoie une matrice diagonale de diagonale d_1,...d_n
```

```
A[1, 0] # terme de la deuxieme ligne, premiere colonne 3
A[0, :] # premiere ligne sous forme de tableau a 1 dimension array([1, 2])
A[:, 1] # deuxieme colonne sous forme de tableau a 1 dimension array([2, 4, 6])
A[:, 1:2]# deuxieme colonne sous forme de matrice colonne
A[:, :1]# premiere colonne colonne sous forme de matrice colonne
array([[2],[4],[6]])
A[[1,2], 1:3] # sous-matrice lignes 2 et 3, colonnes 1 et 2
```

Dans le module `numpy.linalg` :

```
alg.inv(A) # renvoie l'inverse de A
alg.eigvals(A) # renvoie les valeurs propres de A
alg.eig(A)  # renvoie les vecteurs propres de A
alg.solve(A,b) # resout AX=b
np.vdot(u,v) # produit scalaire canonique
```

Exercice 1.1 *Proposer un programme prenant pour paramètre une matrice ou un vecteur et comptant le nombre de terme nul.*

```
def nbzero(A):
    n=0
    for u in A:
        if u==0:
            n=n+1
    return n

def nbzeromatrice(A):
    n=0
    for u in A:
        for v in u:
            if v==0:
                n=n+1
    return n
 # ou encore pour les petits malins
def nbzero_bis(A):
      import numpy as np
      return np.sum(A==np.zeros(np.shape(A)))
```

Exercice 1.2 $A=\begin{pmatrix}2&1&-2\\0&3&0\\1&-1&5\end{pmatrix}$ *et* $B=\begin{pmatrix}1&-1&-1\\-3&3&-3\\-1&1&1\end{pmatrix}$ *on pose*

$$X_{n+2}=\frac{1}{6}AX_{n+1}+\frac{1}{6}BX_n \quad X_0=\begin{pmatrix}3\\0\\-1\end{pmatrix} \quad X_1=\begin{pmatrix}3\\0\\-2\end{pmatrix}.$$

Proposer une fonction prenant pour paramètre n et renvoyant X_n

```
def Xn(n):
      import numpy as np
      A=np.numpy([[2,1,-2],[0,3,0],[1,-1,5]])
      B=np.array([[1,-1,-1],[-3,3,-3],[-1,1,1]])
      X=[np.array([3,0,-1]),np.array([3,0,-2])]
      for i in range(2,n):
            X=X+[(1/6)*np.dot(X[i+1],A.T)+(1/6)*np.dot(X[i],B.T)]]
      return X[n]
```

Exercice 1.3 *Soit $Q\in\mathcal{M}_n(\mathbb{N})$, vérifiant :*

$$\begin{cases}\forall i,j\in\mathbb{N}_*^2 & q_{1,j}=q_{i,1}=1\\ \forall i\in\mathbb{N}^*\setminus\{1\} & q_{i,i}-q_{i-1,i}=1\\ \forall i<j & q_{i,j}=q_{i-1,j}+q_{i,j-i}\\ \forall i>j & q_{i,j}=q_{i-1,j}\end{cases}$$

Proposer un programme (une fonction) prenant pour paramètre n et renvoyant la matrice Q.

```
def Q(n):
    import numpy as np
    Q=np.zeros((n,n))
```

```
    for i in range(n):
         Q[i,0],Q[0,i]=1,1
    for i in range(1,n):

         for j in range(1,i):
             Q[j,i]=Q[i-1,j]+Q[i,j-1]
         for j in range(i,n):
             Q[i,j]=Q[i-1,j]
    return Q
```

2 Simulation d'expérience aléatoire

Pour simuler du hasard et les lois classiques :

```
import numpy.random as rd
```

Les lois discrètes :

```
rd.randint(a,b,n) # une liste de n VA independantes
                          #de loi uniforme discrete sur (a,b-1)
rd.randint(a,b,(n,p)) # un tableau de n*p VA independantes
                           #de loi uniforme discrete sur (a,b-1)
rd.binomial(n,p,N) #une liste de n VA independantes
                           #de loi binomiale de parametre (n,p)
rd.binomial(n,p,(N,P))  # un tableau de n*p VA independantes
rd.geometric(p,N) #une liste de n VA independantes
                            #de loi geometrique de parametre p
rd.geometric(p,(N,P))  # un tableau de n*p VA independantes
rd.poisson(lambda,N) #une liste de n VA independantes
                                  #de loi Poisson de parametre lambda
rd.poisson(lambda,(N,P))  # un tableau de n*p VA independantes
```

Pour simuler du hasard on peut utiliser `rd.random(n)` qui simule une loi uniforme sur $[0,1]$ c'est à dire renvoie un nombre au hasard entre 0 et 1. Plus précisement la probabilité que `rd.random()` soit inférieur à $x \in [0,1]$ est x :

```
rd.random(5)
Out[1]: array([0.56239901, 0.34074058, 0.50393644, 0.91529074, 0.39195047])

rd.random()
Out[2]: 0.9073030730519794
```

Cette fonction permet de simuler un peu tout ce qu'on veut comme par exemple, on simule une loi binomiale de paramètre $n = 20$ et $p = 0.3$ ainsi :

```
import numpy as np

np.sum(rd.random(10)<0.5)
Out[9]: 4
```

Plus généralement :

```
import numpy as np
def binomial(n,p):
                return np.sum(rd.random(n)<p)
```

On peut simuler une VA suivant une loi géométrique

```
def geometrique(p):
        g=1
        lancer=rd.random()
        while lancer>p:
             lancer=rd.random()
            g=g+1
      return g
```

Exercice 2.1 *On dispose de n cartes retournée face cachée, alignées. Chaque tour on choisit au hasard une carte face cachée et on retourne la carte et celle à sa droite ou seulement la carte face cachée si elle est à l'extrémité droite. On s'arrête si toutes les cartes sont retournée face visible. On note N la VA donnant le nombre de tour effectué jusqu'à l'arrêt du jeu. On numérote les cartes de 1 à n de la gauche vers la droite et on pose X_i la VA prenant la valeur 1 si la carte i est face cachée et 0 sinon. Soit $X = (X_1, ..., X_n)$ la chaine de VA X.*

1. *Programmer une fonction `tour(X)` prenant pour paramètre la chaine X correspondant à l'état des cartes à un instant donné et renvoyant la chaine Y correspondant à l'état des cartes à l'instant suivant un tour.*
2. *Programmer une fonction `carte(n)` renvoyant N pour u n donné. Que constatez vous.*
3. *Proposer une représentation graphique de la loi de N.*

1. Programmer une fonction `tour(X)` prenant pour paramètre la chaine X correspondant à l'état des cartes à un instant donné et renvoyant la chaine Y correspondant à l'état des cartes à l'instant suivant un tour.

```
def tour(X):
     c=len(X)
     listecarte=[]
     for i in range(c):
            if X[i]==1:
               listecarte.append(i)
      n=len(listecarte)
      if n==0:
             return X
      j=rd.randint(n)
      if j==c-1:
            X[j]=0
            return X
      X[j],X[j+1]=1-X[j],1-X[j+1]
      return X
```

2. Programmer une fonction `carte(n)` renvoyant N pour un n donné. Que constatez vous.

```
def carte(n):
      X=[1]*n
      N=0
      while sum(X)!=0:
               X=tour(X)
               N=N+1
      return N
```

3. Proposer une représentation graphique de la loi de N.

3 Manipulation de listes et Compression de texte

Les listes et arrangements :

```
np.arange(a,b,delta) # une liste de nombre entre a et b avec un pas de delta
np.linspace(a,b,N) # une liste de N nombres reparti regulierement entre a et b
rang(n) # une liste d'entier entre 0 et n-1
range(p,n) # une liste d'entier entre 1 et n-1
range(p,n,k)# une liste d'entier de p a n-1 avec un pas de k
```

Pour la lecture et l'utilisation des matrices on fera attention à la nuimerotation qui commence en 0 :

```
A[1, 0] # terme de la deuxieme ligne, premiere colonne 3
A[0, :] # premiere ligne sous forme de tableau a 1 dimension array([1, 2])
A[0, :].shape
(2,)
A[0:1, :] # premiere ligne sous forme de matrice ligne array([[1, 2]])
A[0:1, :].shape
(1, 2)
A[:, 1] # deuxieme colonne sous forme de tableau a 1 dimension array([2, 4, 6])
A[:, 1:2] # deuxieme colonne sous forme de matrice colonne
array([[2],[4],[6]])
A[1:3, 0:2] # sous-matrice lignes 2 et 3, colonnes 1 et 2
```

Opération sur les listes ou les matrices :

```
np.sum(A) # somme des elements de A
np.min()# min des elements de A
np.max()# max des elements de A
np.mean()# moyenne des elements de A
np.cumsum()# somme partielle des elements de A
np.median()# mediane des elements de A
np.var()# variance des elements de A
np.std()# ecart type des elements de A
```

On peut aussi le faire pour chaque colonne ou chaque ligne :

```
np.sum(A,0) # par colonne
np.sum(A,1) # par ligne
```

Des commandes de tris :

```
L.sort() # tri par ordre croissant
L.reverse() #  change l'ordre
min(L) # ressort le min
max(a) # je vous laisse deviner
L.insert(i,a)# insert le terme a en position i
del L[i] # efface le terme en position i
L[i:j] # extrait la sous liste d'indice i a j-1
L.remove(a)# retirer l'element a de L
L.count(i)# indique le nombre de terme de L de valeur i
```

L'objectif est de compresser, puis décompresser un texte (qu'on imaginera être une suite de nombres), efficacement et rapidement. On sera donc amener à utiliser les listes et pratiquer des opérations sur ces même listes.

Un petit rappel sur la complexité :
Le temps d'exécution $T(f)$ d'une fonction f est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc.) nécessaire au calcul de f. Lorsque ce temps d'exécution dépend d'un paramètre n, il sera noté $Tn(f)$. On dit que la fonction f s'exécute : en temps $O(n^a)$, s'il existe $K > 0$ tel que pour tout n, $T(f) \leq Kn^a$ Dans cet exercice, il sera question de l'algorithme de Burrows-Wheeler qui compresse très efficacement des données textuelles.

Exercice 3.1 *Le texte d'entrée à compresser sera un tableau* `t` *contenant des entiers compris entre* 0 *et* 255 *inclus.*

1. *La compression par redondance compresse un texte d'entrée qui possède des répétitions consécutives de lettres (ou d'entiers dans notre cas). Dans un premier temps, on calcule les fréquences d'apparition de chaque entier dans le texte d'entrée. Puis on compresse le texte.*
   (a) *Écrire la fonction* `occurrences(t)` *qui prend en argument un tableau d'entrée* `t` *de longueur* n *; et qui retourne un tableau* r *de taille* 256 *tel que* $r[i]$ *est le nombre d'occurrences de* i *dans* t *pour* $0 \leq i < 256$.
   (b) *Appliquer (tester le programme) le sur des exemples concrets.*
   (c) *Écrire la fontion* `min(t)` *qui prend en argument le tableau* `t` *de longueur* n *; et qui retourne le plus petit entier de l'intervalle* $[0, 255]$ *qui apparaît le moins souvent dans le tableau* `t`*. (Le nombre d'occurrences de cet entier peut être nul)*
   (d) *Tester le programme.*
2. *L'entier* $min(t, n)$ *servira de marqueur. On note* m *ce marqueur et, pour simplifier, on suppose que son nombre d'occurrences est nul. Donc* `r[m]` $= 0$ *quand* $r =$ `occurrences(t)`*.*
   La compression par redondance du texte `t` *fonctionne comme suit : toute répétition maximale contigüe d'une lettre où* $t[i] = t[i+1] = \cdots = t[j] = k$ *est codée par les trois entiers* $m, (j-i), k$ *; toute apparition unique d'une lettre* k *est codée par cette même lettre. Le premier terme de la liste est le marqueur* m
   Par exemple, si `t` $= [0, 0, 3, 2, 3, 3, 3, 3, 3, 3, 5]$*. Le marqueur est donc* 1 *car* 1 *n'apparaît pas dans ce tableau. Le texte compressé est alors* $[m, m, 2-1, 0, 3, 2, m, 5-4, 3, 5] = [1, 1, 1, 0, 3, 2, 1, 5, 3, 5]$
   (a) *Écrire la fonction* `tailleCodage(t)` *qui prend comme argument le tableau* t *et calcule la taille* n *du texte compressé c'est à dire la longueur de la liste compréssé.*
   (b) *Appliquer sur d'exemple précédant.*
   (c) *Écrire la fonction* `codage(t)` *qui prend comme paramètre le tableau* t *et retourne un tableau d'entiers représentant le texte compressé.*
   (d) *Pour pouvoir décoder un texte ainsi compressé, il suffit de connaître le marqueur utilisé. Or ce marqueur est le premier entier du texte compressé. Proposer un programme* `Decodage(r)`*, renvoyant le texte d'origine à partir du texte compressé.*
3. *Le codage par redondance n'est efficace que si le texte présente de nombreuses répétitions consécutives de lettres. Ce n'est évidemment pas le cas pour un texte pris au hasard. La transformation de Burrows-Wheeler est une transformation qui, à partir d'un texte donné, produit un autre texte contenant exactement les mêmes lettres mais dans un autre ordre où les répétitions de lettres ont tendance à être contigües. Cette transformation est bijective. Considérons par exemple le texte d'entrée* `concours`*. Pour simplifier la présentation, nous utilisons ici des caractères pour le tableau d'entrée. Cependant, dans les programmes, on considère toujours (comme dans la première partie) que le texte d'entrée est un tableau d'entiers compris entre* 0 *et* 255 *inclus. Le principe de la transformation suit les trois étapes suivantes :*
   — *On regarde toutes les rotations du texte. Dans notre cas, il y en a* 8 *qui sont :*

   ```
   concours oncoursc ncoursco courscon oursconc ursconco rsconcou sconcour
   ```

   — *On trie ces rotations par ordre lexicographique (l'ordre du dictionnaire).*

   ```
   concours courscon ncoursco oncoursc oursconc rsconcou sconcour ursconco
   ```

   — *Le texte résultant est formé par toutes les dernières lettres des mots dans l'ordre précédent, soit snoccuro dans l'exemple, ainsi que de l'indice de la lettre dans ce texte résultant qui est la première lettre du texte original, soit* 3 *dans notre exemple. On appelle cet entier la clé de la transformation.*
   On remarque que les deux c du texte de départ se retrouvent côte à côte après la transformation. En effet, comme le tri des rotations regroupe les mêmes lettres sur la première colonne, cela conduit à rapprocher aussi les lettres de la dernière colonne qui les précèdent dans le texte d'entrée.
   (a) *Proposer une fonction* `Insertion(x,i,r)` *qui renvoie la liste après avoir insérer la lettre* x *dans la liste* r *en* i*-ème position.*

(b) *Proposer une fonction récursive* `rot(t)` *qui renvoie la liste des textes distincts obtenu par rotation toutes les rotations distinctes possible classer dans l'ordre lexicographique (on pourra utiliser la fonction de tri* `sort()`*.*

(c) *Programmer la fonction* `codageBW(t)`*, qui renvoie la liste* `t'` *de taille* $n+1$ *dont le dernier terme contient la clé. Il restera plus qu'à compresser par redondance.*

4. *Pour décompresser, les ennuis commences. :*

(a) *On programme la fonction* `tricar(t')` *qui renvoie liste tri, triée dans l'ordre croissant de* `t'` *(sans la clé).*

(b) *Voici la procédure pour construire la liste* `t"` *(magique de décompression)*
- *A chaque caractère* u *de t' on associe le caractère* v *de même position de la liste* `tri`
- *A chaque caractère* v *de tri on associe le même caractère de même rang de la liste* `t'`
- *On initialise avec le premier caractère* $u0$ *de* `tri`*, comme premier terme de* `t"`
- *Si* $v0$ *est le caractère associé à* $u0$ *de* `tri`*, le second caractère de* `t"` *est le caractère de même rang de la liste* `t'`*, noté* $u1$*, et ainsi de suite.*
- *La liste* `t` *est obtenu en effectuant autant de permutation que la clé.*

Programmer le decodage.

1. (a)
```
def occurences(t):
    r=[0]*256
    for x in t:
        r[x]+=1
    return r
```

(b)
```
t=rd.randint(0,255,10000)
occurences(t)
......
```

(c)
```
def m(t):
    r=occurences(t)
    i=0
    for j in range(255):
        if r[i]<r[j]:
            i=j
    return i
```

2. (a)
```
def tailleCodage(t):
    taille=1
    j=-1
    u=1
    for i in t:
        if i!=j:
            taille+=u
            u=1
        else:
            u=3
    taille +=u
    return taille
```

(b) Tester sur l'exemple :

(c)
```
def codage(t):
    m=m(t)
    T=[m]
```

```
    j=-1
    u=1
    for i in t:
         if i!=j:
             if  u==1:
                   T.append(i)
                    j=i
              else:
                     T=T+[m,u,i]
                      j=i
                      u=1
          else:
               u+=1
    if u!=1:
         T=T+[m,u,i]
    return T
```

```
def Decodage(r):
    m=r[0]
    T=[]
    i=1
    for i <=len(r):
          if r[i]!=m:
               T.append(r[i])
               i+=1
           else:
                 T=T+[r[i+2]   ]*r[u[i+1]]
                 i+=3

    return T
```

(3) (a)

```
def insertion(x,i,r):
      return r[:i-1]+[x]+r[i-1:]
```

(b)

```
def rot(t):
     T=[t]
     for i in range(len(t)):
          T.append(T[i][1:]+t[i])
     return T.sort()
```

(c)

```
def codageBW(t):
      tprime=[]
      cle=t[0]
      T=rot(t)
      for u in T:
           tprime.append(u[-1])
      tprime.append(cle)
      return tprime
```

4. (a)

```
def tricar(t)
      return t[:-1].sort()
```

```
def decodage(t):
      tri=tricar(t)
      u=tri[0]
      cle=t[-1]
      del t[-1]
      tseconde=[u]
      j=0
      while len(tri)>0:
                i=0
                while t[i]!=u:
                          i+=1
                u=tri[i]
                del tri[j]
                j=i
                del t[i]
                tseconde.append(u)
         for i in range(len(tseconde)):
                  if tseconde[i]==cle:
                       return tseconde[i:]+tseconde[:i]
```

(b) **Exercice 3.2** *Pours les* $\frac{5}{2}$

En base 3*, les entiers* $0, 1, 2, 3, 4, 5, 6, 7, 8$ *sont représentés par* $00, 01, 02, 10, 11, 12, 20, 21, 22$*. Le chiffre de poids fort de bc est b ; le chiffre de poids faible est c.*

1. (a) *Proposer une fonction* `poids(bc)` *prenant comme paramètre le chiffre bc de poids respectifs b et c et renvoyant le uplet* `(b,c)`*.*

   (b) *Écrire la fonction* `entier(b,c)` *renvoyant l'entier compris entre* 0 *et* 8 *qui s'écrit bc en base* 3*.*

   (c) *Soit x un entier vérifiant* $0 \leq x \leq 8$*. Ècrire une fonction* `poidsFort(x)` *retournant le chiffre de poids fort de* `x` *en base* 3*.*

   (d) *Écrire la fonction* `poidsFaible(x)` *retournant le chiffre de poids faible de* `x`*.*

2. *Dans ce problème, les textes sont représentés en représentation ternaire. Un savant russe nous a convaincus de la pertinence de ce choix plus compact que la représentation binaire. Un texte est rangé dans un tableau* `t` *(une liste de liste) de N caractères vérifiant* `t[i]` *est un entier appartenent à* $\{0, 1, 2\}$ *pour tout i vérifiant* $0 \leq i < N - 1$ *; par ailleurs* `t[N-1]`$= X > 2$ *(le dernier caractère n'est pas ternaire). On suppose* $N \geq 1$*.*

   Quelques définitions sont nécessaires :
   - *la chaîne de caractères de longueur l démarrant en i est la liste* `[t[i],t[i+1],...t[i+l-1]]`*.*
   - *On dira que deux chaînes* `[t[i],t[i+1],...t[i+l-1]]` *et* `[t[j],t[j+1],...t[j+l'-1]]`*sont égales si* $l = l'$ *et* $t[i + k] = t[j + k]$ *pour* $0 \leq k < l$*.*

   (a) *Proposer une fonction* `Conversion(t)` *prenant une chaine d'entiers* `t` *compris entre* 0 *et* 8 *et renvoyant la une chaine de nombre converti en ternaire.*

   (b) *Après avoir importer la bibliothèque* `numpy.random`*, tester la fonction conversion sur une chaine bien choisit de longueur* 10*.*

   (c) *Écrire une fonction* `longueurMotif(t,i,j,m)` *qui retourne, avec une complexité linéaire, c-a-d d'ordre* $O(N)$*, la plus grande longueur l d'une chaîne de* `t` *démarrant en i égale à une autre chaîne de* `t` *démarrant en j. En outre, cette longueur doit vérifier* $l \leq m$ *(on prendra garde de ne pas dépasser la longueur de la chaine* `t`*...). Tester le programme sur une chaine de taille* 50*.*

   (d) *On suppose* $i < j$*. Écrire une fonction* `longueurMotifMax(t, i, j, m)` *qui retourne, avec une complxité quadratique, c-a-d* $O(N^2)$ *, la plus grande longueur l d'une chaîne démarrant en* $i+k$ *égale à une chaîne démarrant en j pour* $0 \leq k < m$*. En outre, on exige* $i + k < j$ *et* $l \leq m$*. Tester le programme sur une chaine de taille* 50*.*

(e) *Modifier la fonction précédente pour obtenir la fonction* `motifMax(t,i,j,m)` *qui renvoie les variable global* `A,L,C`*, avec une complexité quadratique : L la plus grande longueur l d'une chaîne démarrant en* $i+k$ *égale à une chaîne démarrant en* j *pour* $0 \leq k < m$ *;* `A` *la valeur de* k *pour lequel* $i+k$ *est l'indice de départ de cette chaîne de longueur maximale ;* `C` *le caractère suivant cette chaîne à partir de* j *dans* t*. À nouveau, cette longueur doit vérifier* $l \leq m$*. Et on a* $i+k<j$*. On pourra compléter :*

```
def motifmax(t,i,j,m):
    global A,L,C
    ..................
    return A,L,C
```

3. *La méthode de compression de Ziv et Lempel, adoptée dans les commandes zip ou gzip, consiste à repérer les motifs maximaux déjà rencontrés dans un texte et à indiquer pour chacun d'eux le triplet* `(A,L,C)` *calculé dans la question précédente entre toute paire d'indices* i *et* j*. Pour mesurer le facteur de compression, nous utilisons le même codage pour ces triplets que pour les caractères du texte, c'est-à-dire le système ternaire dans ce problème.*

   (a) *Écrire une fonction* `imprimerTriplet(A,L,C)` *qui renvoie les arguments* `A, L, C` *sous forme de cinq chiffres consécutifs, les deux caractères ternaires de* `A`*, puis les deux caractères ternaires de* `L`*, puis le chiffre* `C`*, en imposant* $0 \leq A < 9$*,* $0 \leq L < 9$ *et* $0 \leq C \leq 9$*.*

   (b) *On suppose à présent que* `t` *contient un long texte ternaire commençant par* 9 *caractères* `0` *; en outre,* `t` *finit par un caractère* `x` *spécial (*$x > 2$*). On déplace sur ce texte une fenêtre de longueur* 18*. Au début, cette fenêtre est alignée à gauche sur le début du tableau, et on pose* $j = 9$*. En régime de croisière, la dixième case de la fenêtre correspond à l'entrée* j *du tableau* `t`*. On recherche, dans la partie gauche de la fenêtre, la chaîne de longueur* l *maximale vérifiant* $l < 9$ *et égale à une chaîne de caractères démarrant en* j*. On imprime, grâce à la fonction* `imprimerTriplet`*, le triplet* `(A, L, C )` *donné par* `motifMax`*. Puis, on recentre la fenêtre sur le caractère suivant le caractère d'arrêt* `C`*. Ce processus continue jusqu'au bout du tableau* `t` *comme indiqué par la figure. Ainsi pour le texte suivant, on obtient les décompositions de chacune des lignes, soit au total le facteur de compression* 30/37 *qui serait nettement meilleur dans une base supérieure à* 3 *et si la taille de la fenêtre était plus grande que* 18 *(Il y a en effet* 30 *caractères dans le résultat et* 37 *dans le texte d'entrée). Écrire une fonction* `compresser(t)` *qui renvoie le texte compressé par la méthode précédente.*

   (c) *Pour la décompression, on produit d'abord* 9 *caractères* 0*. On considère ensuite tous les triplets* `(A, L, C )` *représentés par* 5 *caractères ternaires consécutifs et on recrée la chaîne originale jusqu'au dernier triplet dont la composante* `C` *n'est pas comprise entre* 0 *et* 2*. . Écrire une fonction* `décompresser(tc)` *qui prend un texte ternaire* `tc` *comme paramètre et qui renvoie le texte décompressé correspondant.*

```
import numpy as np
import numpy.random as rd
def poids(bc):
    return bc//10,bc%10
def entier(b,c):
    return 3*b+c
def poidsfort(x):
    return x//3
def poidsfaible(x):
    return x%3
def convert(t):
    T=[]
    for x in t:
        T.append(poidsfort(x)*10+poidsfaible(x))
    return T
# t=rd.randint(0,8,10)
```

```
# convert(t)
def longueurMotif(t,i,j,m):
    l=0
    m=np.min([len(t)-i,m,len(t)-j])
    while l<=m:
        if t[i+l]!=t[j+l]:
            return l
        l+=1
    return l
def longueurMotifmax(t,i,j,m):
    assert j>i
    M=0
    for k in range(j-i):
        l=longueurMotif(t, i+k, j, m)
        if l>M:
            M=l
    return M
def motifmax(t,i,j,m):
    #global A,L,C
    assert j>i
    M,K=0
    for k in range(j-i):
        l=longueurMotif(t, i+k, j, m)
        if l>M:
            K,M=k,l
    A,L,C=M,K,t[j+1]
    return A,L,C
```