

# 1 Introduction

La recherche d'**algorithmes de compression** est ancienne en informatique, avec deux buts visés :

- **gagner de l'espace** en réduisant l'espace de stockage;
- **gagner du temps** en réduisant le temps de transfert des données.

Les progrès technique ont permis d'améliorer notablement les capacités de stockage et les débits des liaisons mais avec le développement d'Internet, Les données numériques sont de plus en plus nombreuses, , on parle de Big Data. Ces données jouent un rôle de plus en plus important pour l'analyse et la prise de décision (data mining), le divertissement (flux video . . . ) ou l'entraînement des algorithmes d'apprentissage (machine learning). Il donc est important de disposer de bons algorithmes de compression.

En informatique, toutes les données sont représentées au plus bas niveau sous forme de bits, huit bits formant un octet. Un **algorithme de compression** prend en entrée un flux d'octets de taille  $B$  et renvoie un flux d'octets compressés de taille  $C(B)$ . Il est d'autant plus performant que le **ratio de compression**  $\frac{C(B)}{B}$  est petit. Évidemment, un tel algorithme est utile seulement si on connaît un **algorithme de décompression** capable de restituer le flux d'octets initial à partir du flux compressé.

On distingue deux catégories d'algorithmes de compression :

- Les **algorithmes de compression sans perte** d'information que nous étudierons dans ce mini-projet et qui exploitent des redondances dans les données sources pour réduire l'espace (images, textes . . . )
- Les **algorithmes de compression avec perte** pour les informations traitées par nos sens : une perte d'information est tolérée si elle ne réduit pas de façon significative la qualité de la perception (d'une image, d'un son . . . )

Plan du projet :

- **Partie A** : développer des outils de conversion de textes, d'images, d'entiers en flux d'octets.
- **Partie B** : découverte de l'algorithme de compression RLE.
- **Partie C** : découverte de l'algorithme de compression LZW.

# 2 Cahier des charges

1. Les programmes de chaque partie doivent être rassemblés dans des fichiers différents nommés `partieA.py`, `partieB.py` et `partieC.py`.
2. Chaque fonction de `partieA.py`, `partieB.py` ou `partieC.py` doit passer le test inclus correspondant dans les programmes de tests `test_partieA.py`, `test_partieB.py` ou `test_partieC.py` fournis sinon cela doit être mentionné dans le code source sous forme de commentaires.
3. Les réponses aux questions qui ne nécessitent pas de code doivent être fournies dans le code source sous forme de commentaire en les préfixant par le numéro de la question.
4. Chaque fonction doit être documentée avec une `docstring`.
5. Les parties les moins évidentes du code doivent être commentées de façon pertinente.
6. Un code client permettant de tester le programme ou les fonctions principales doit être fourni.

### 3 Partie A : boîte à outils

L'objectif de cette partie est de nous outiller avec des fonctions de conversions (dans les deux sens) de textes (type `str`) ou d'images (avec le module `PIL`) en flux d'octets (de type `bytes`) qui seront compressés puis décompressés par les algorithmes présentés en parties B et C.

1. Extraire l'archive `DM-Compression.zip` dans un répertoire du même nom. Elle contient :
  - les trois squelettes de code `partieA.py`, `partieB.py`, `partieC.py`
  - les trois fichiers de tests `test_partieA.py`, `test_partieB.py` et `test_partieC.py`.
  - des ressources (textes, images ...) utilisées pour tester le code
2. Renommer le répertoire en `DM-Compression_Eleve1_Eleve2` avec les noms des membres du groupe séparés par des tirets bas (surtout pas d'espace). Par la suite on appellera ce répertoire le répertoire de base du projet.
3. Exécuter le script `partieA.py`. Si le module `PI` n'est pas installé, une erreur s'affiche, il faut alors l'installer en suivant la documentation en ligne sur :

<https://pillow.readthedocs.io/en/stable/installation.html>

```
1 from PIL import Image
```

4. Compléter la fonction `index_premiere_occurrence(element, tab)` en respectant la spécification donnée dans la docstring.

Vérifier si la fonction passe le test unitaire `index_premiere_occurrence` dans `test-partieA.py`.

```
def index_premiere_occurrence(element, tab):  
    """  
    Parametres :  
        element de type quelconque (le même que les éléments de tab)  
        tab un tableau d'éléments de même type  
    Valeur renvoyée:  
        l'indice de la première occurrence de element dans tab  
    """
```

5. Compléter la fonction `hex_to_decimal(hexadecimal)` en respectant la spécification donnée dans la docstring.

Vérifier si la fonction passe le test unitaire `test_hex_to_decimal` dans `test_partieA.py`.

```
def hex_to_decimal(hexadecimal):  
    """  
    Parametres :  
        hexadecimal de type str représentant un entier en base 16  
    Valeur renvoyée:  
        représentation décimale de hexadecimal sous forme d'entier de  
        type int  
    """
```

6. Compléter la fonction `decimal_to_hex(n)` en respectant la spécification donnée dans la docstring. Vérifier si la fonction passe le test unitaire `test_decimal_to_hex` dans `test-partieA.py`.

```
def decimal_to_hex(n):
    """
    Paramètres :
        n de type int
    Valeur renvoyée:
        représentation de n en base 16 sous forme de chaîne de caractères
        on rajoute un 0 à gauche si un seul chiffre en base 16
    """
    rep = ''
    #à compléter
```

7. `partieA.py` contient ensuite trois fonctions de conversion entre les types `str` des chaînes de caractères et `bytes` des flux/séquences d'octets qui seront manipulés par nos algorithmes de compression.

```
def str_to_bytes(chaine, encodage = 'utf8'):
    """
    Paramètres :
        chaine de type str, une chaîne de caractères
        encodage un paramètre de type str fixant l'encodage, par défaut utf8
    Valeur renvoyée :
        un flux d'octets de type bytes obtenu par encodage de chaine
    """
    return chaine.encode(encoding = encodage)


def bytes_to_str(flux, encodage = 'utf8'):
    """
    Paramètres :
        flux d'octets de type bytes
        encodage un paramètre de type str fixant l'encodage, par défaut utf8
    Valeur renvoyée :
        une chaîne de caractères de type str obtenue par décodage de flux
    """
    return flux.decode(encoding = encodage)

def hex_to_bytes(hexadecimal):
    """
    Paramètres :
        hexadecimal de type str représentant un nombre à 2 chiffres en base 16
    Valeur renvoyée :
        un octet de type bytes représentant le même nombre qu' hexadecimal
    """
```

```
return bytes.fromhex(hexadecimal)
```

Pour comprendre leur fonctionnement, on donne ci-dessous quelques exemples, disponibles également dans l'interpréteur en ligne **Basthon**. On peut remarquer que les caractères imprimables de la table ASCII (ordinal Unicode inférieur à 128) sont représentés tels quels dans le type `bytes`.

Pourquoi n'est-ce pas le cas pour le caractère accentué `é` dans l'exemple ci-dessous?

 *Pour simplifier, nous travaillerons dans ce DM uniquement avec des chaînes de caractères de la table ASCII codés sur un octet.*

```
>>> [str_to_bytes(c) for c in ['1', 'a', 'A', 'é']]
[b'1', b'a', b'A', b'\xc3\xa9']
>>> [bytes_to_str(b) for b in [b'1', b'a', b'A', b'\xc3\xa9']]
['1', 'a', 'A', 'é']
>>> [hex_to_bytes(h) for h in ['00', '06', '0B', 'A1', 'FF']]
[b'\x00', b'\x06', b'\x0b', b'\xa1', b'\xff']
```

### Complément

Les dernières fonctions outils fournies dans `partieA.py` permettent de convertir un fichier représentant une image en niveaux de gris (pixel sur un octet) en un flux d'octets de type `bytes` et réciproquement de convertir un flux d'octets en image (en précisant sa largeur et sa hauteur).

On utilisera ces fonctions comme des boîtes noires. Pour comprendre leur fonctionnement on donne ci-dessous un exemple d'exécution sur une image de  $2 \times 2$  pixels représentant un damier : on extrait le flux d'octets de l'image source, on le transforme en remplaçant les octets par leur complémentaire à 255 puis on convertit ce flux de sortie en image.

Le test unitaire `test_image` fourni dans `test_partieA.py` permet de reproduire cet exemple.

```
In [7]: img = fichier_to_image('damier_2x2.png')

In [8]: img
Out[8]: <PIL.Image.Image image mode=L size=2x2 at 0x7FA724517340>

In [9]: img_bytes = image_to_bytes(img)

In [10]: img_bytes
Out[10]: b'\xff\x00\x00\xff'

In [11]: img_bytes_sortie = b''.join([(255 - img_bytes[k]).to_bytes(1,
    byteorder='little') for k in range(len(img_bytes))])

In [12]: img_bytes_sortie
Out[12]: b'\x00\xff\xff\x00'

In [13]: img_sortie = bytes_to_image(img_bytes_sortie, img.width, img.height
    )

In [14]: img_sortie
Out[14]: <PIL.Image.Image image mode=L size=2x2 at 0x7FA7240C4C40>
```



## 4 Partie B : compression RLE

\_\_\_\_\_ Complément \_\_\_\_\_

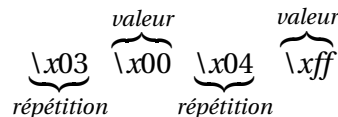
### Méthode

**Run Length Encoding** est un algorithme efficace pour compresser des données contenant de longues séquences de valeurs répétées.

La chaîne de caractères "ABBBAABBB" peut ainsi être compressée en "1A3B2A4B".

Le principe est simple : on représente chaque séquence de caractères identiques par sa longueur suivie du caractère répété.

En pratique, on travaille sur des séquences d'octets, ainsi la séquence de 7 octets en notation hexadécimale : `\x00\x00\x00\xff\xff\xff\xff` va être compressée en une séquence de 4 octets :



On fixe également une longueur maximale pour les répétitions : dans notre mini-projet, on choisit de les coder sur 1 octet soit 255 valeurs consécutives.

\_\_\_\_\_ Fin de complément \_\_\_\_\_

1. Compresser avec l'algorithme **Run Length Encoding** la séquence d'octets ci-dessous en notation hexadécimale. Donner la séquence compressée en notation décimale également.

`\x00\x00\x0a\x0a\x0a\x0a\xff\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff`

2. Décompresser la séquence d'octets ci-dessous compressée avec l'algorithme **Run Length Encoding** :

`\x02\xb1\x0a\x00\x01\xfc\x0f\x0a`

3. Ouvrir le script `partieB.py`. Quel est le rôle de la première instruction ci-dessous ?

```
from partieA import *
```

---

*Complément*

---

Avant de traiter les questions suivantes, il faut se familiariser avec les objets de type `bytes`. Consulter le tutoriel [Exemples\\_Operations\\_Bytes.pdf](#) et ce [notebook Basthon](#).

---

*Fin de complément*

---

4. Compléter la fonction `decompresse_rle(flux)` en respectant la spécification donnée dans la docstring. Vérifier si la fonction passe le test unitaire `test_decompresse_rle` dans `test_partieB.py`.

```
def decompresse_rle(flux_comprime):
    """
    Paramètre :
        flux_comprime de type bytes est un flux d'octets compressé avec
        l'algorithme RLE
    Valeur renvoyée :
        flux_sortie de type bytes qui est la décompression de
        flux_comprime
    """
    flux_sortie = b''
    "à compléter"
    return flux_sortie
```

5. Compléter la fonction `comprime_rle(flux)` en respectant la spécification donnée dans la docstring. Vérifier si la fonction passe le test unitaire `test_comprime_rle` dans `test_partieB.py`.

```
def compresse_rle(flux):
    """
    Paramètre :
        flux de type bytes est un flux d'octets
    Valeur renvoyée :
        flux_comprime de type bytes qui est la compression de flux avec
        l'algorithme RLE
    """
    courant = flux[0:1] #pour avoir un bytes
    compteur = 1
    flux_comprime = b'' #de type bytes
    for k in range(1, len(flux)):
        "à compléter"
    return flux_comprime
```

6. Compléter la fonction `ratio_rle(flux, flux_comprime)` en respectant la spécification donnée dans la docstring.

```
def ratio_rle(flux_comprime, flux):
    """
    Paramètre :
        flux de type bytes est un flux d'octets
        flux_comprime de type bytes est un flux d'octets
    Valeur renvoyée :
```

```

    un nombre de type float représentant le ratio entre flux et
    flux_comprese
"""
"à compléter"

```

7. Compléter le script `partieB.py` avec un code client permettant de tester l'exemple fourni dans la page Wikipedia sur la compression **Run Length Encoding**.

Compression RLE d'un texte

Source : WWWWWWBWWWWWWWWWWWBBBWWWWWWWWWWWWWWWBWWWWWWWWW

Flux d'octets : b'WWWWWWWBWWWWWWWWWWWBBBWWWWWWWWWWWWWWWBWWWWWWWWW'

Flux d'octets compressé : b'\x0cW\x01B\x0eW\x03B\x17W\x01B\x0bW'

Sortie : WWWWWWBWWWWWWWWWWWBBBWWWWWWWWWWWWWWWBWWWWWWWWW

Ratio de compression : 0.215385

8. Compléter le script `partieB.py` avec un code client permettant de compresser l'image en niveau de gris enregistrée dans le fichier `lena.png` : extraction du flux d'octets représentant l'image, compression de ce flux, affichage du ratio, décompression et conversion du flux de sortie en image. On utilisera les fonctions `fichier_to_image`, `image_to_bytes` et `bytes_to_image` définies dans `partieA.py`.

```


lena_img = fichier_to_image('lena.png')
flux = image_to_bytes(lena_img )
#à compléter
lena_img_decompresse = bytes_to_image(flux_sortie, lena_img.width,
    lena_img.height)
lena_img_decompresse.save('lena-decompresse.png')

```

## 5 Partie C (facultative, en bonus) : compression LZW

### \_\_\_\_\_ Complément \_\_\_\_\_

L'algorithme de compression **LZW** est un algorithme de compression de données sans perte inventé par les chercheurs israéliens Abraham Lempel et Jacob Ziv en 1978 puis amélioré par Terry Welch en 1984.

 Cet algorithme est utilisé pour compresser des flux d'octets extraits d'images ou de textes mais pour mieux visualiser son déroulement nous considérerons une **situation simplifiée** où l'algorithme compressé une chaîne de caractères ASCII (ordinal Unicode inférieur à 128, **codés sur un octet chacun**) que nous appellerons *texte source*.

Le principe est la construction d'une **table des symboles** qui associe les caractères ou des séquences du caractère du texte source à des codes. Le texte source est alors compressé en un flux d'octets représentés par les codes associés.

Pour implémenter la **table des symboles** en Python, nous utiliserons la structure de données **dictionnaire** qui est présentée dans le tutoriel [Exemples\\_Dictionnaires\\_Bytes.pdf](#) et ce **notebook Basthon**.

La **table des symboles** peut être reconstruite lors de la phase de décompression, il suffit donc de transmettre le flux d'octets compressé et l'alphabet dans lequel le texte (ou l'image ...) source a été composé.

### \_\_\_\_\_ Fin de complément \_\_\_\_\_

1. Ouvrir les fichiers `partieC.py` et `test_partieC.py`.

2. Consulter le tutoriel [Exemples\\_Dictionnaires\\_Bytes.pdf](#) et ce [notebook Basthon](#) pour se familiariser avec la structure de données dictionnaires (type dict).

### Complément

Voici une description de l'algorithme de compression :

#### Méthode

- On initialise la table de symboles avec les caractères de l'alphabet (les caractères de la table ASCII dans notre situation simplifiée) en leur associant par exemple leur ordinal Unicode entre 0 et 127.

```
table = {chr(k) : decimal_to_hex(k) for k in range(128)}
```

- On initialise un flux d'octets vide `flux_comprime` pour recevoir la compression du texte source.
- Dans une boucle, on parcourt le texte source du premier caractère au dernier et lors de chaque itération on exécute le bloc d'instructions suivant :
  - on recherche la plus longue chaîne `s` de caractères codée dans la table de symboles, qui est un préfixe des caractères qui n'ont pas encore été parcourus ;
  - on ajoute le code numérique associé à `s` à `flux_comprime` ;
  - on lit le caractère `c` qui suit `s` dans le texte source, on concatène `s` et `c` et on enregistre `s + c` dans la table de symboles en lui associant un nouveau code numérique.
  - on reprend l'itération suivante à partir du caractère `c`.

On peut dérouler un exemple d'exécution sur la page Wikipedia :

[https://fr.wikipedia.org/wiki/Lempel-Ziv-Welch#/media/Fichier:Imaged1\\_lzw.svg](https://fr.wikipedia.org/wiki/Lempel-Ziv-Welch#/media/Fichier:Imaged1_lzw.svg).

Pour bien comprendre l'algorithme de compression, nous allons dérouler un autre exemple : on compresse le texte source ABABABA et on utilise un alphabet ASCII de 128 caractères enregistrés initialement dans la table des symboles avec des codes entre 0 et 127. Pour la table des symboles, on ne mentionne pas l'alphabet, uniquement les séquences de caractères ajoutées à chaque itération (codées à partir de  $80^{16}$  en base 16 soit 128 en décimal).

#### • Étape 1

- Texte source : ABABABA
- Table des symboles : 

Caractères	AB
Code	80
- Flux compressé d'octets en hexadécimal : 41

#### • Étape 2

- Texte source : ABABABA
- Table des symboles : 

Caractères	AB	BA
Code	80	81
- Flux compressé d'octets en hexadécimal : 41 42

#### • Étape 3



– Texte source : ABABABA

– Table des symboles :

Caractères	AB	BA	ABA
Code	80	81	82

– Flux compressé d'octets en hexadécimal : 41 42 80

• Étape 4

– Texte source : ABABABA

– Table des symboles (inchangée, pas de caractère suivant) :

Caractères	AB	BA	ABA
Code	80	81	82

– Flux compressé d'octets en hexadécimal : 41 42 80 82

————— Fin de complément —————

9. Compléter la fonction `recherche_plus_long_prefixe(chaine, pos_initiale, table)` en respectant la spécification donnée dans la docstring.

Vérifier si la fonction passe le test unitaire `test_recherche_plus_long_prefixe` dans `test_partieC.py`.

```
def recherche_plus_long_prefixe(chaine, pos_initiale, table):
    """
    Paramètres :
        chaine de type str
        pos un entier avec 0 <= pos < len(chaine)
        table de type dict
    Valeurs renvoyées :
        pos de type int
        prefixe de type str
    Postcondition :
        chaine[pos_initiale:pos] est le plus long préfixe de chaine[
            pos_initiale:]
        qui est une clef dans table
    """
    prefixe = ''
    pos = pos_initiale
    # à compléter
    return pos, prefixe
```

10. Compléter dans `partieC.py`, la fonction `compression_lzw(chaine, table_max = 256)` en respectant la spécification donnée dans la docstring.

On limite le nombre de séquences de caractères à 256 (de 0 à 255) afin que chaque code numérique puisse être codé sur un octet.

Vérifier si la fonction passe le test unitaire `test_compression_lzw` dans `test_partieC.py`.

```
def compression_lzw(chaine, table_max = 256, debug = False):
    """
    Paramètres :
        chaine de type str
        table_max un entier de type int
    Valeurs renvoyées :
```

```
flux_compresse de type bytes
table de type dict
Postcondition :
    table contient au plus table_max clefs
    flux_compresse est la compression LZW de chaine
    avec table pour table de symboles
"""
table = {chr(k) : decimal_to_hex(k) for k in range(128)}
index_suivant_table = 128
flux_compresse = b''
pos = 0
while pos < len(chaine):
    # à compléter
    flux_compresse = flux_compresse + hex_to_bytes(table[prefixe])
if debug: #pour débogage
    print("Mode débogage, affichage de la table de symboles")
    print(table)
return flux_compresse
```

On donne ci-dessous un exemple d'exécution avec un affichage partiel de la table de symboles en mode débogage.

```
>>> compression_lzw('ABABABA', table_max = 256, debug = True)
Mode débogage, affichage de la table de symboles
{'\x00': '00', '\x01': '01', ..., '0': '30', '1': '31', '2': '32', '3':
 '33', ..., '@': '40', 'A': '41', 'B': '42', ..., '[': '5B', ..., 'a':
 '61', 'b': '62', ..., 'z': '7A', '{': '7B', '|': '7C', '}': '7D', '~
 ': '7E', '\x7f': '7F', 'AB': '80', 'BA': '81', 'ABA': '82'}
Flux d'octets compressé : b'AB\x80\x82'
```

### Complément


Voici une description de l'algorithme de décompression. On rappelle que la compression a transformé un texte source constitué de caractères ASCII en un flux d'octets codés en hexadécimal, chacun représentant un caractère ou une séquence de caractères ASCII présents dans le texte source. Par abus de langage, on peut identifier octet et code numérique dans cette phase de décompression, puisque chaque code produit par l'algorithme de compression tient sur un octet.

### Méthode

- On initialise une table de symboles inverse de celle de la phase de compression, qui associe aux codes, les entiers entre 0 et 127 notés en hexadécimal, les caractères ASCII dont ils sont les ordinaux.

```
table_inv = {decimal_to_hex(k) : chr(k) for k in range(128)}
```

- On initialise une chaîne de caractères vide sortie pour recevoir le décodage du flux d'octets représentant le texte source compressé.
- Dans une boucle, on parcourt le flux d'octets du premier octet au dernier et lors de chaque itération on exécute le bloc d'instructions suivant :

- on lit l'octet courant dans le flux;
- on écrit dans la chaîne `sortie` la séquence de caractère(s) `s` associée à l'octet lu dans `table_inv`;
- on lit l'octet suivant (s'il existe) dans le flux et on extrait le premier caractère `c` de la séquence de caractère(s) associée;
- on ajoute dans `table_inv` l'association entre le premier code disponible (on commence à 128 en décimal et  $\overline{80}^{16}$  en hexadécimal) et la séquence de caractères `s + c`.
-  il y a un cas particulier : si l'octet suivant ne figure pas encore dans `table_inv` alors il s'agit de l'octet qu'on doit ajouter dans la table à cette étape et donc son premier caractère est le premier caractère de la séquence associée à l'octet courant (celui qu'on lit).

On peut dérouler un exemple d'exécution sur la page Wikipedia :

[https://fr.wikipedia.org/wiki/Lempel-Ziv-Welch#/media/Fichier:Imaged1\\_lzw.svg](https://fr.wikipedia.org/wiki/Lempel-Ziv-Welch#/media/Fichier:Imaged1_lzw.svg).

Pour bien comprendre l'algorithme décompression, nous allons dérouler un autre exemple : on décompresse le flux d'octets 41 42 80 82 obtenu précédemment par compression du texte source ABABABA.

Tout d'abord on initialise notre table des symboles inverse en associant aux entiers entre 0 et 127, en hexadécimal, les caractères de notre alphabet, la table ASCII.

Pour la table des symboles, on ne mentionne que les codes supérieurs ou égaux à 128 soit  $\overline{80}^{16}$  en hexadécimal des séquences de caractères ajoutées à chaque itération.

#### • Étape 1

- Flux d'octets : 41 42 80 82.

L'octet lu est 41, `table_inv[41]` renvoie le caractère A dont 41 est l'ordinal en base 16. On ajoute dans la table la séquence de caractères AB constituée du caractère qui vient d'être décodé et du premier caractère de la séquence associée dans `table_inv` à l'octet suivant (ici 42 qui est l'ordinal en base 16 de B).

- Table des symboles inverse : 

Code	80
Caractères	AB

- Chaîne de caractères en sortie : A

#### • Étape 2

- Flux d'octets : 41 42 80 82.

L'octet lu est 42, `table_inv[42]` renvoie le caractère B dont 42 est l'ordinal en base 16. On ajoute dans la table la séquence de caractères BA constituée du caractère qui vient d'être décodé et du premier caractère de la séquence associée dans `table_inv` à l'octet suivant (ici 80 qui est associé à AB).

- Table des symboles inverse : 

Code	80	81
Caractères	AB	BA

- Chaîne de caractères en sortie : AB

#### • Étape 3

- Flux d'octets : 41 42 80 82.

L'octet lu est 80, `table_inv[80]` renvoie la séquence de caractères AB. On lit l'octet suivant 82 pour récupérer le premier caractère de la séquence associée, mais il n'est pas encore dans la table. Il s'agit donc de l'octet qu'on doit ajouter à cette étape dont le premier caractère est donc le premier de la séquence qu'on vient de décoder, c'est-à-dire A. Ainsi on ajoute dans la table la concaténation AB + A soit ABA associée à 82.

- Table des symboles inverse :
- |            |    |    |     |
|------------|----|----|-----|
| Code       | 80 | 81 | 82  |
| Caractères | AB | BA | ABA |
- Chaîne de caractères en sortie : ABAB

• Étape 4

- Flux d'octets : 41 42 80 82.  
L'octet lu est 82, `table_inv[82]` renvoie la séquence de caractères ABA. Il n'existe pas d'octet suivant donc on n'ajoute rien dans la table, c'est la dernière étape de l'algorithme.
- Table des symboles inverse :
- |            |    |    |     |
|------------|----|----|-----|
| Code       | 80 | 81 | 82  |
| Caractères | AB | BA | ABA |
- Chaîne de caractères en sortie : ABABABA

————— Fin de complément —————

11. Compléter dans `partieC.py`, la fonction `decompression_lzw(flux, table_max = 256, debug = True)` en respectant la spécification donnée dans la docstring.

On limite le nombre de séquences de caractères à 256 (de 0 à 255) afin que chaque code numérique puisse être codé sur un octet.

Vérifier si la fonction passe le test unitaire `test_decompression_lzw` dans `test_partieC.py`.

```
def decompression_lzw(flux, table_max = 256, debug = True):
    table_inv = { decimal_to_hex(k) : chr(k) for k in range(128)}
    index_suivant_table = 128
    sortie = ''
    pos = 0
    while pos < len(flux):
        #flux de type bytes donc flux[pos] est un entier !
        code_courant = decimal_to_hex(flux[pos])
        #à compléter
    if debug: #pour débogage
        print("Mode débogage, affichage de la table de symboles inversée")
        print(table_inv)
    return sortie
```

12. Décrire le format de fichiers FASTA en quelques lignes.

On a fourni dans `partieC.py` une fonction permettant d'extraire le code génétique contenu dans un fichier FASTA : une très longue chaîne de caractères construit sur l'alphabet A, T, G et C pour Adénine, Thymine, Guanine et Cytosine.

L'archive du mini-projet contient également un fichier texte `sequence_e-coli.fasta` contenant le code génétique d'**Escherichia coli** une bactérie de l'intestin.

```
def fasta_to_str(fichier_fasta):
    """
    Paramètre :
        fichier_fasta de type str un chemin vers un fichier fasta
    Valeur renvoyée :
```

```

output de type str représentant le code génétique contenu dans
le fichier
"""
f = open(fichier_fasta)
# saut de la première ligne
f.readline()
output = ''
for ligne in f:
    #on enlève le saut de ligne
    output = output + ligne.rstrip()
f.close()
return output

```

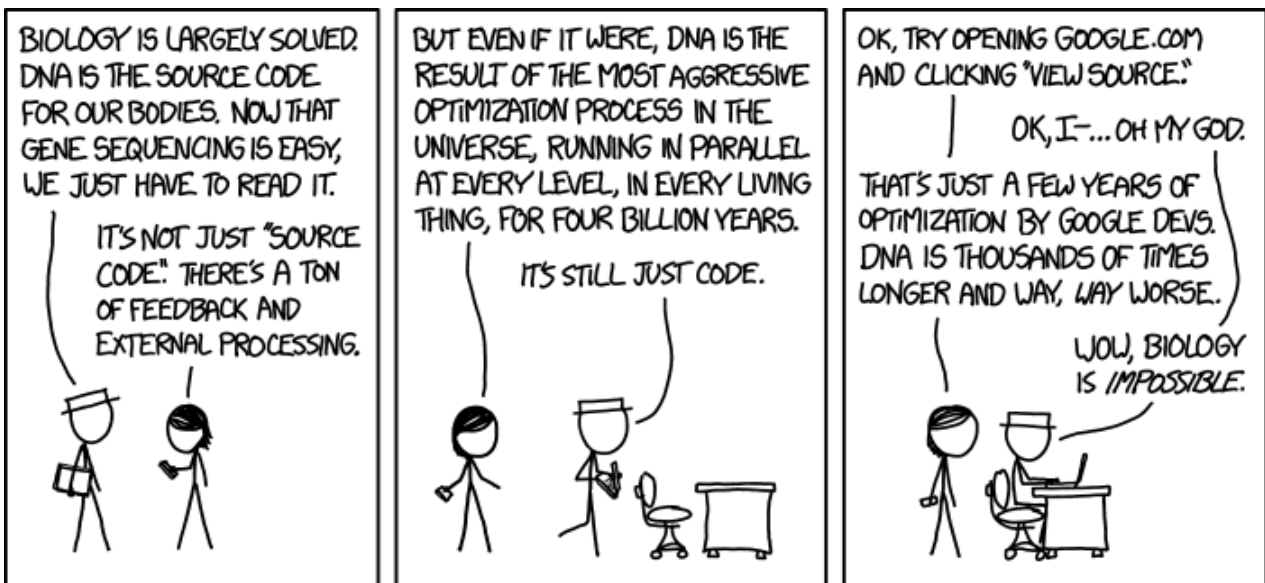
Compléter partieC.py avec un code client permettant d'extraire le code génétique contenu dans `sequence_e-coli.fasta`, de le compresser en un flux d'octets avec `compression_lzw`, de décompresser avec `decompression_lzw`, de vérifier que la correspondance est correcte pour les 10 000 premiers caractères et d'afficher le ratio de compression.

Mesurer le temps d'exécution des phases de compression et de décompression avec la fonction `perf_counter` du module `time`.

```

Test de compression / décompression sur le code génétique d'Escheria
Ecoli
Compression en 197.441295 s
Décompression en 1.723819 s
Ratio de compression : 0.33421
Comparaison des 10 000 premiers caractères : True

```



Source et explication : [https://www.explainxkcd.com/wiki/index.php/1605:\\_DNA](https://www.explainxkcd.com/wiki/index.php/1605:_DNA)