TP 10 - Compression d'un flux (deuxième partie)

L'objectif est de continuer à développer le module `compresseur.py`.

1 Rappels et compléments

Pour rappel, la phase de compression peut être décomposée en différentes étapes :

1. calcul des statistiques à partir du flux source (considéré comme étant composé d'octets),
2. construction de l'arbre de Huffman à partir des statistiques,
3. calcul du code binaire de chaque octet,
4. production des octets compréssés dans un flux destination à partir des données du flux source et des codes de chaque octet.

Plus exactement dans le flux destination il doit y avoir :

— un code d'identification qui permettra lors d'une demande de décompression de vérifier le type du fichier,
— les statistiques du fichier source,
— la longueur du fichier source,
— les données compressées.

Ce qui se synthétise par la figure 1.

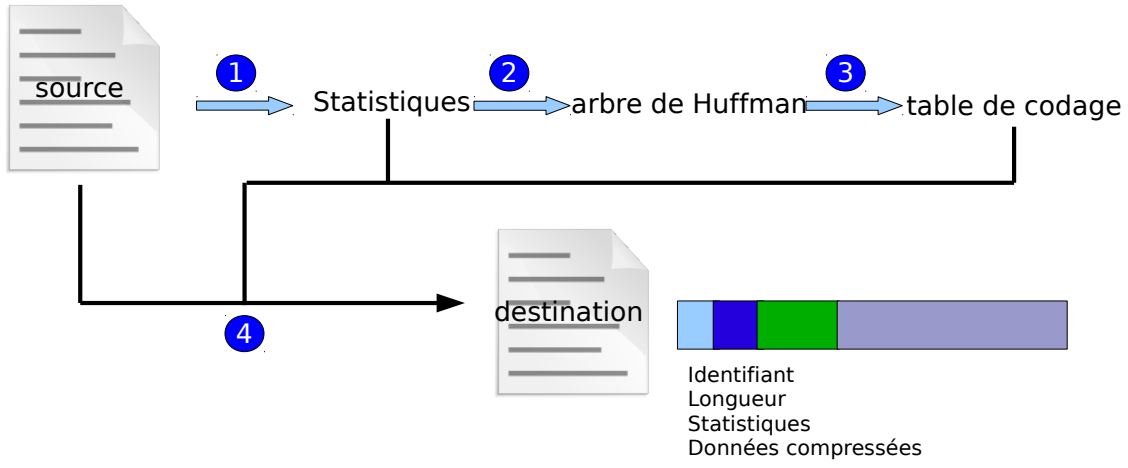
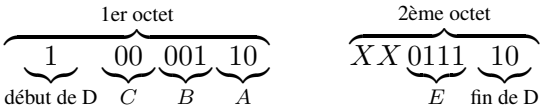


FIGURE 1 – Les phases de la compression

Il est à noter que ceci est le fonctionnement général du compresseur, et qu'il y a deux cas particuliers, lorsque la source à compresser est composée :

1. d'aucun octet ;
2. du même octet répéter n fois.

La construction des octets des données compressées se fait de droite à gauche et l'utilisation des codes binaires de gauche à droite. Par exemple, pour reprendre le codage du précédent TP (présenté par le tableau 1), les octets représentant les données compressées à partir des données sources $ABCDE$ sont :



...avec les X qui représentent des bits non significatifs.

En effet le code binaire du premier octet à compresser, le 'A', étant 01, on met le bit le plus à droite du premier octet des données compressées à 0 et le deuxième (en partant de la droite) à 1. Ensuite le code binaire de deuxième octet à compresser, le 'B', étant 100, on met le 3ème bit du premier octet des données compressées (toujours en partant de la droite) à 1, le 4ème à 0 et le 5ème à 0, etc. Comme le code binaire du 4ème octet des données à compresser, le 'D', est sur 3 bits et qu'il reste uniquement 1 bit de disponible sur le 1er octet des données compressées, le 2ème bit de son code binaire, le 0, est mis sur le 1er bit (le plus à droite) du 2ème octet des données compressées, etc.

Caractère	Code binaire
A	01_2
B	100_2
C	00_2
D	101_2
E	1110_2
F	1111_2
G	110_2

TABLE 1 – Un codage

Pour finir, nous faisons le choix que les bits non utilisés du dernier octet des données compressées doivent être à 0. Ainsi les octets des données compressées à partir des octets sources $ABCEF$ sont 10000110 et 00011101 (les précédents X sont à 0).

2 Code python

Dans le module `compresseur.py`, vous devez coder la fonction suivante qui enregistre toutes les informations nécessaires dans le flux destination :

— `compresser(destination: io.RawIOBase, source: io.RawIOBase)`

```
def compresser(destination: io.RawIOBase,
               source: io.RawIOBase,
               nb_octets_pour_serialisation_des_int: int=4,
               ordre_pour_serialisation_des_int='big') -> None:
```

Tels que :

— l'identifiant de fichier sera composé de 2 octets, le premiers 52 (en hexadécimal 34) et le deuxième 50 (en hexadécimal 32) ;

— le troisième octet de flux compresser vaudra :

  — 0 s'il n'y pas de données à compresser (fichier vide)

— 1 si les données à compresser contiennent qu'un seul octet, dans ce cas après la longueur des données à compresser il y aura l'octet en question
— 2 sinon

Annexe

Obtention d'octets

Il va donc falloir enregistrer des octets (`bytes`) dans le flux `destination`. Pour cela vous utiliserez la méthode `write` qui prend en paramètre un objet de type `bytes` (séquence d'octets). Vous serez amené à obtenir un objet de type `bytes` en :

— utilisant une constante, par exemple `b"\x00\x01\x02"` ;
— créant une instance de `bytes` à partir d'une séquence de `int` représentant des octets, donc compris entre `0` et `255`, par exemple `bytes([0,1,2])` ;
— créant une instance de `bytes` à partir de la représentation d'une valeur d'un `int`. Il faut dans ce cas utiliser la méthode `to_bytes` de la classe `int`, tel que le premier paramètre est le nombre d'octets pour représenter l'`int` et le second indique l'ordre de prise en compte des octets (`"big"` ou `"little"`) pour calculer la valeur de l'`int`, par exemple :
`int(258).to_bytes(4,"big")`

Manipulation des bits d'un octet (naturel compris entre 0 et 255)

Pour construire les octets des données à compresser on va devoir fixer les bits à 0 ou à 1 de ces octets en partant de la droite. Le plus simple est de travailler avec des puissances de 2 et l'opérateur `ou` bit à bit, notée |. Par exemple le code python suivant permet d'obtenir l'octet ayant la représentation binaire 01101011 :

```
>>> octet = 0
>>> octet = octet | 2**0 | 2**1 | 2**3 | 2**5 | 2**6
>>> f"{octet:>08b}"
'01101011'
```