TP 9 - Compression d'un flux (première partie)

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

1 Bibliographie : Présentation de la méthode

On se propose dans cette partie de présenter le principe de la compression de Huffman. Afin d'être plus clair, nous allons voir comment coder un texte (composé de caractères). Le même raisonnement peut être appliqué pour compresser des données binaires (composées d'octets).

1.1 Principe

Le principe de cette méthode est de remplacer l'utilisation d'un code [1] à longueur fixe (par exemple 8 bits) par un code à longueur variable : un caractère souvent présent dans un document source sera alors codé à l'aide d'un code plus court qu'un caractère n'apparaissant que quelques fois [2].

Pour réaliser cet algorithme nous avons besoin d'un arbre de Huffman dont les feuilles sont les caractères présents dans le document source. Le chemin permettant d'atteindre ces feuilles est le code correspondant (avec ici comme protocole 0 pour l'accès à un sous-arbre gauche et 1 pour l'accès à un sous-arbre droit).

Par exemple si le texte source est "BACFGABDDACEACG", on obtient l'arbre binaire présenté par la figure 1a. Ce qui donne le code à longueur variable présenté par le tableau 1b.

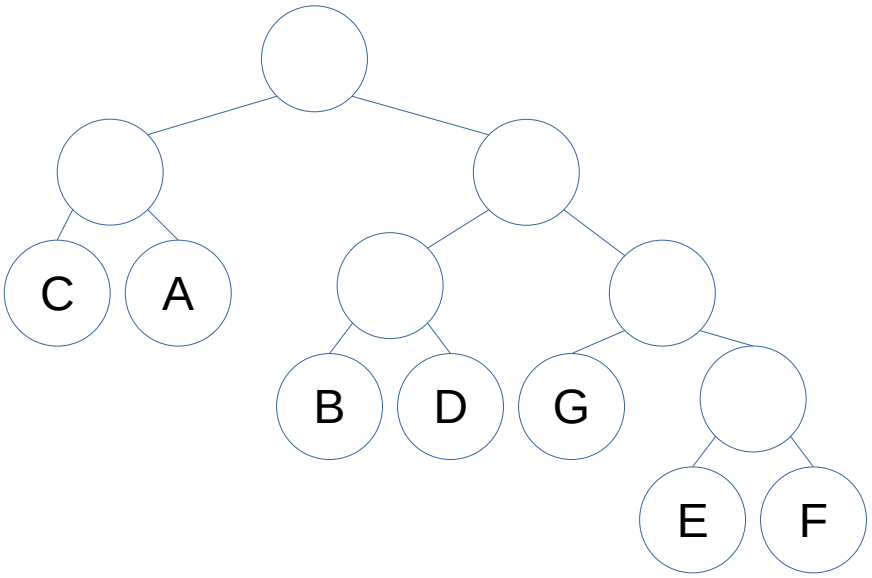
1.2 Méthode pour construire l'arbre de Huffman

La méthode pour construire cet arbre est la suivante :

1. On commence par construire autant d'arbres qu'il y a de caractères dans le texte source en ajoutant la fréquence d'apparition. Dans notre exemple cela donne :

2. On construit alors une file de priorité d'arbres. On y insère ces arbres, dans l'ordre croissant de leur code ASCII (le 'A' sera inséré avant le 'B'), en considérant qu'un arbre a_1 est plus prioritaire qu'un autre arbre a_2, lorsque la pondération de la racine de a_1 est plus petite que celle de a_2. Dans notre exemple on obtient alors la file de priorité suivante :

1. Un code est une suite de bits
2. C'est ce qui est appliqué dans le langage Morse

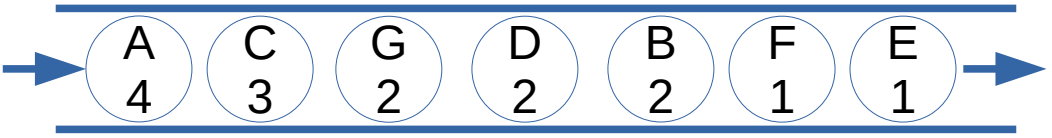


(a) Arbre de Huffman

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

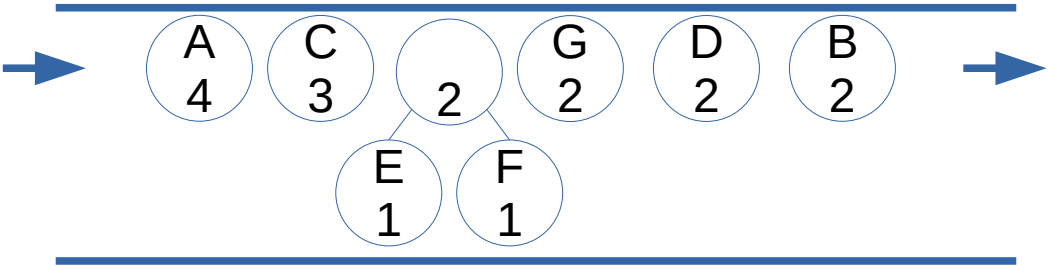
(b) Table de codage

FIGURE 1 – Arbre de Huffman et table de codage correspondante

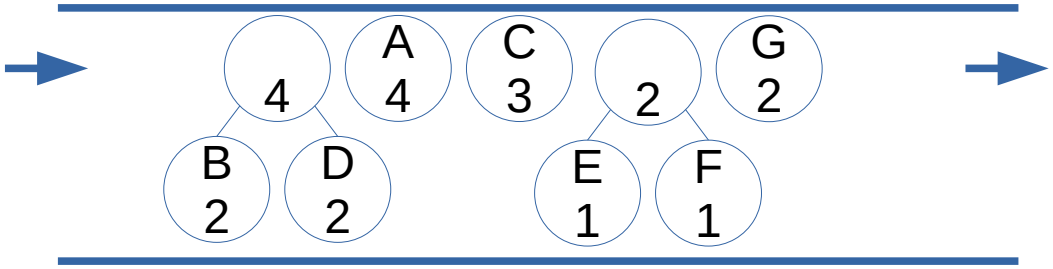


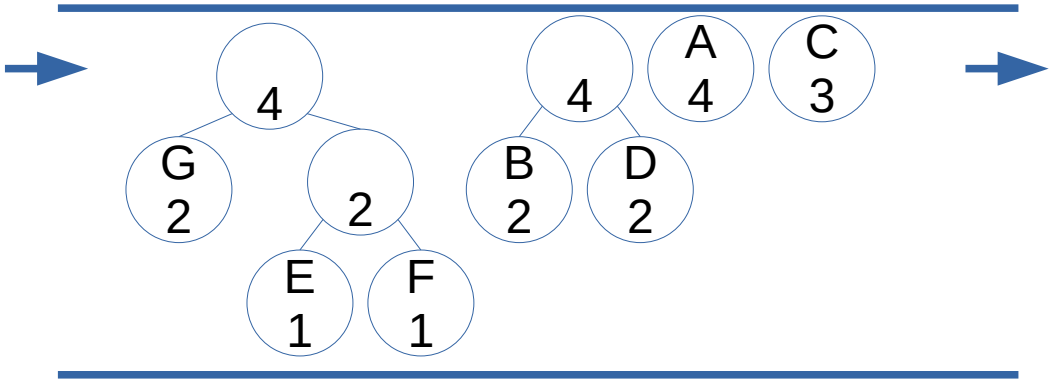
3. S'il y a au moins deux arbres dans la file de priorité :
   — on défile ces deux arbres
   — on en forme un nouveau dont :
      — la pondération de la racine sera la somme des pondérations des deux racines de ces fils,
      — le sous-arbre gauche est le premier arbre défilé,
      — le sous-arbre droit est le deuxième arbre défilé,
   — on insère ce nouvel arbre dans la file.

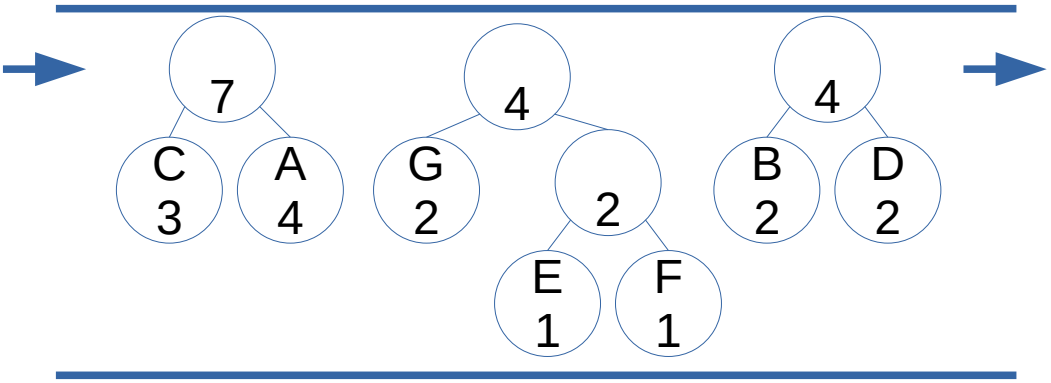
   Dans notre exemple on obtient alors la file de priorité suivante :

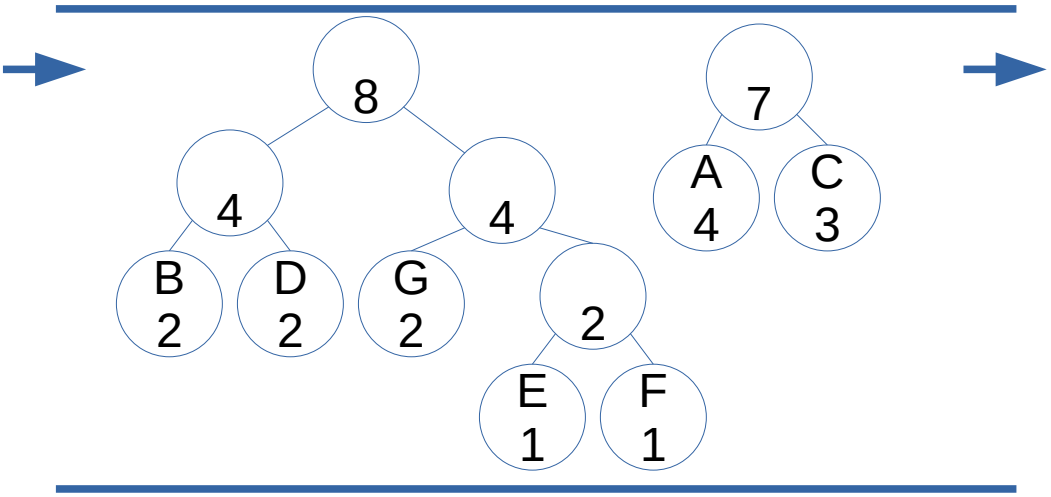


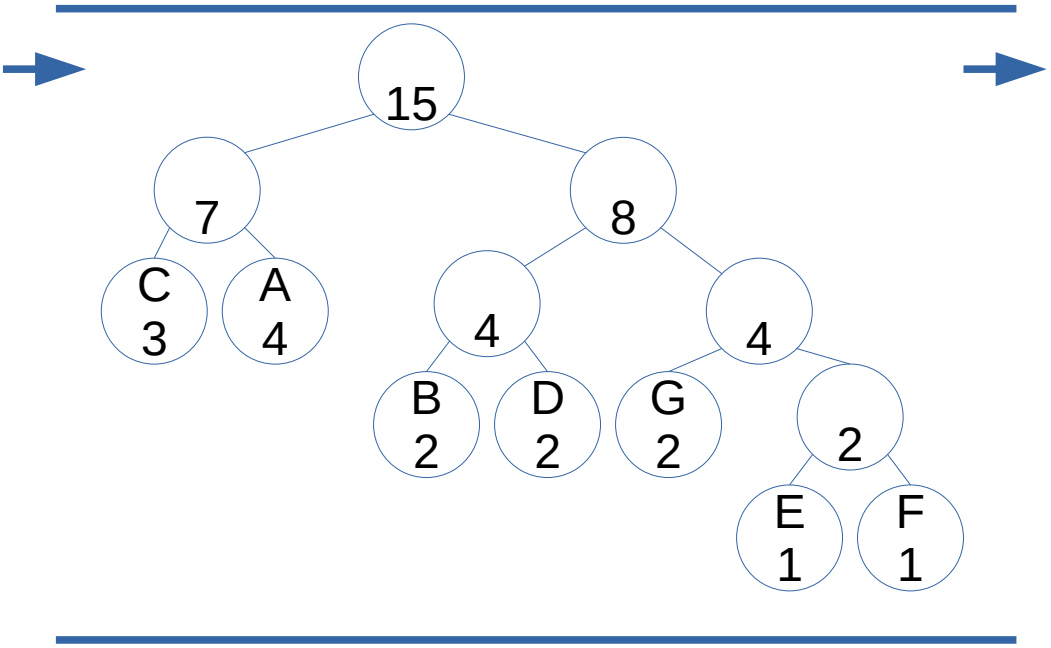
4. On réitère l'opération jusqu'à ce qu'il n'y ait plus qu'un seul arbre






7
C
3
A
4
4
G
2
2
E
1
F
1
4
B
2
D
2


8
4
B
2
D
2
4
G
2
2
E
1
F
1
7
A
4
C
3



1.3 Compression

Ainsi le texte “BACFGABDDACEACG” peut être codé de la façon suivante :

$$\underbrace{100}_{B}\ \underbrace{01}_{A}\ \underbrace{00}_{C}\ \underbrace{1111}_{F}\ \underbrace{110}_{G}\ \underbrace{01}_{A}\ \underbrace{100}_{B}\ \underbrace{101}_{D}\ \underbrace{101}_{D}\ \underbrace{01}_{A}\ \underbrace{00}_{C}\ \underbrace{1110}_{E}\ \underbrace{01}_{A}\ \underbrace{00}_{C}\ \underbrace{110}_{G}$$

Il faut maintenant découper cette suite de bits en suite de 8 bits [3] :

10001001 11111001 10010110 10100111 00100110

Il faut maintenant représenter cette suite de 8 bits en suite d’octets. Très souvent on « remplit » les octets en commençant par le bit de poids faible. Ainsi les 8 premiers bits de la suite de bits précédentes (10001001) peut être représentée par l’octet en binaire 10010001, soit 145 en décimal et 91 en hexadécimal. On obtient alors le suite d’octets suivante :

en binaire 10010001 10011111 01101001 11100101 01100100

en décimal 145 159 105 229 100

en hexadécimal 91 $9F$ 69 $E5$ 64

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. Ces données seront constituées :

3. Le fait que le dernier code binaire remplisse entièrement le dernier octet est pure coïncidence.

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

Ce qui se synthétise par la figure 2.

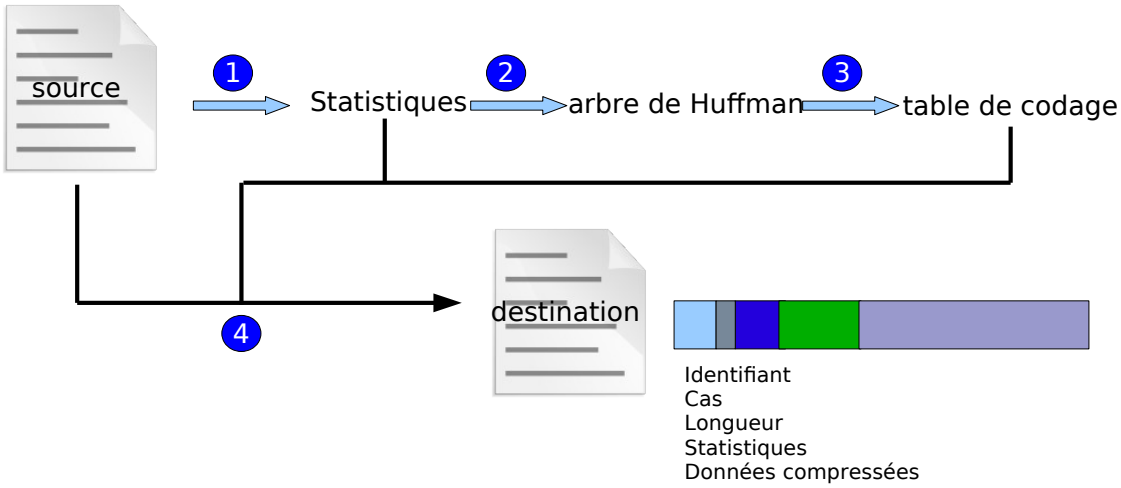


FIGURE 2 – Les phases de la compression

2 Code python

Nous allons réaliser dans le cadre de ce TP les étapes 1 à 3. Vous devez coder les fonctions suivantes :

— `statistiques(source: io.BufferedReader) -> (Compteur, int)`
qui calcule le nombre d'octets d'un flux binaire (deuxième élément du tuple retourné) et le nombre d'occurrences de chaque octet (premier élément du tuple retourné)
— `arbre_de_huffman(stat: Compteur) -> ArbreHuffman`
qui construit un arbre de Huffman à partir d'un compteur (cf. explications ci dessus en enfilant initialement les octets en ordre croissant)
— `codes_binaire(abr: ArbreHuffman) -> Dict[int, CodeBinaire]`
qui construit le code binaire de chaque octet
Des tests unitaires vous permettront de tester votre code.