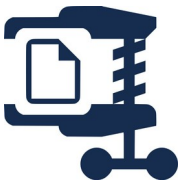
Le codage de Huffman



Ce sujet propose d'étudier une méthode de compression de données inventée par David Albert Huffman en 1952, qui permet de réduire la longueur du codage d'un alphabet et qui repose sur la création d'un arbre binaire.

1. Principe du codage

On appelle alphabet l'ensemble des symboles (caractères) composant la donnée de départ à compresser. Dans la suite, nous utiliserons un alphabet composé seulement des 8 lettres A, B, C, D, E, F, G et H.

On cherche à coder chaque lettre de cet alphabet par une séquence de chiffres binaires.

Q1. **Indiquer** le nombre de bits nécessaires pour coder chacune des 8 lettres de l'alphabet.
Q2. **Calculer** la longueur en octets d'un message de 1 000 caractères construit sur cet alphabet.
Q3. **Proposer** un code de taille fixe pour chaque caractère de l'alphabet de 8 lettres.

On considère maintenant le codage suivant, la longueur du code de chaque caractère étant variable.

Lettre	A	B	C	D	E	F	G	H
Code	10	001	000	1100	01	1101	1110	1111

Ce type de code est dit préfixe, ce qui signifie qu'aucun code n'est le préfixe d'un autre (le code de A est 10 et aucun autre code ne commence par 10, le code de B est 001 et aucun autre code ne commence par 001). Cette propriété permet de séparer les caractères de manière non ambiguë.

Q4. En utilisant la table précédente, **donner** le code du message : CACHE.
Q5. **Déterminer** le message correspondant au code 001101100111001.

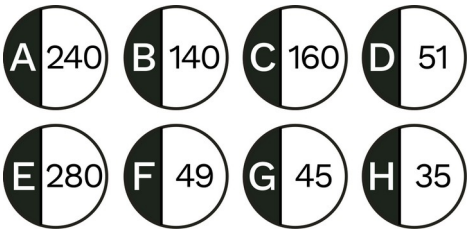
Dans un texte, chacun des 8 caractères a un nombre d'apparitions différent. Cela est résumé dans le tableau suivant, construit à partir d'un texte de 1 000 caractères.

Lettre	A	B	C	D	E	F	G	H
Nombre	240	140	160	51	280	49	45	35

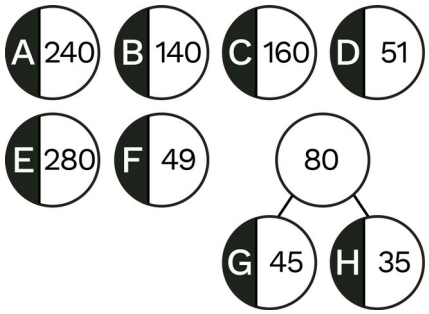
Q6. En utilisant le code de taille fixe proposé à la question Q2, **calculer** la longueur en bits du message contenant les 1 000 caractères énumérés dans le tableau précédent.
Q7. En utilisant le code de la question Q3, **calculer** la longueur du même message en bits. **Calculer** le taux de compression en % ainsi obtenu.

2.Construction du codage de Huffman

L'objectif du codage de Huffman est de trouver le codage proposé en Q3 qui minimise la taille en nombre de bits du message codé en se basant sur le nombre d'apparition de chaque caractère (un caractère qui apparaît souvent aura un code plutôt court).
Pour déterminer le code optimal, on considère 8 arbres, chacun réduit à une racine, contenant le symbole et son nombre d'apparitions.



Puis on fusionne les deux arbres contenant les plus petits nombres d'apparitions (valeur inscrite sur la racine), et on affecte À ce nouvel arbre la somme des nombres d'apparitions de ses deux sous-arbres. Lors de la fusion des deux arbres, le choix de mettre l'un ou l'autre à gauche n'a pas d'importance. Nous choisissons ici de mettre le plus fréquent à gauche (s'il y a un cas d'égalité, nous faisons un choix arbitraire).

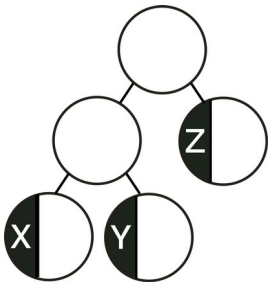


On recommence jusqu'à ce qu'il n'y ait plus qu'un seul arbre.

Q8. **Déterminer** le nombre d'étapes (combien de fusions d'arbres) nécessaires pour que cette algorithme se termine.

Q9. En suivant l'algorithme précédent, **construire** l'arbre de Huffman.

Le code à affecter à chaque lettre est déterminé par sa position dans l'arbre. Précisément, le code d'un symbole de l'alphabet décrit le chemin de la racine à la feuille qui le contient : un 0 indique qu'on descend par le fils gauche et un 1 indique qu'on descend par le fils droit.
Dans le cas de l'arbre ci-contre, le code de X est 00 (deux fois à gauche), le code de Y est 01, et celui de Z est 1.
Sur chaque arête de l'arbre construit à la question Q2, inscrire 0 ou 1 selon que l'arête joint un fils gauche ou un fils droit.



Q10. **En déduire** le code de F .

3.Programmation de la construction du code d'Huffman

Le code suivant (page 3) permet, à partir d'un fichier nommé texte.txt, de construire l'arbre de Huffman puis un dictionnaire qui associe a chaque caractère du fichier d'entrée son code sous forme d'une séquence de bits (liste de 0 et de 1).

Q11. **Compléter** le code en indiquant ce qui manque dans les zones rectangulaires puis **tester** la solution sur ordinateur

```
import bisect
class ArbreHuffman:
  '''Défini un arbre d'Huffmann.
  Paramètres : lettre -> Lettre de la racine de l'arbre (str)
               nbocc -> nombre d'occurrence de la lettre (int)
               g -> sous arbre gauche (None par defaut)____
               d -> sous arbre droit (None par defaut) ____|-> Feuille
  '''
  def __init__(self, lettre:str, nbocc:int, g=None, d=None):
        self.lettre = lettre
        self.nbocc = nbocc
        self.gauche = g
        self.droite = d

  def __lt__ (self, other) -> bool :
  # Un arbre A est strictement inférieur à un arbre B si le nombre
  d'occurrences
  #indiqué dans A est strictement supérieur à celui de B
        return self.nbocc > other.nbocc

  def est_feuille(self) -> bool :
        return

def parcours(arbre:ArbreHuffman, chemin_en_cours:list, dico:dict):
  '''Parcourt en profondeur l'arbre d'Huffman et attribut le code
  aux différents noeuds suivant leur position dans l'arbre
  Paramètres : arbre : Arbre d'Huffman parcouru en profondeur
  chemin en cours : code en cours sur le noeud
  dico : dictionnaire lettres : codes
  '''
  if arbre is None: return
  elif arbre.est_feuille(): dico[arbre.lettre] =
  else:
        parcours(arbre.gauche, chemin_en_cours + [0], dico)

def fusionne(gauche:ArbreHuffman, droite:ArbreHuffman) -> ArbreHuffman:
  '''Fusionne gauche et droite dans un arbre '''
  nbocc_total=

  return ArbreHuffman(
```

```
def compte_occurrences(texte: str) -> dict:
    '''Renvoie un dictionnaire avec chaque caractere du texte comme clé et
    le nombre d'apparition de ce caractere dans le texte en valeur
    >>> compte occurrences ("AABCECA" )
    {"Av: 3, "BU: 1, "C": 2, "E": 1}
    '''
    occ = {}
    for caract in texte:
        if caract not in occ:
            
        else :
            occ[caract] += 1
    return 

def construit_liste_arbres(texte: str) -> list:
    ''' Renvoie une liste d'arbres de Huffman, chacun réduit à une feuille
    >>> construit_liste_arbres('AAABB')
    [|A:3|,|B:2|]
    '''
    dic_occurrences = compte_occurrences(texte)
    liste_arbres = []
    for 


    return 

def codage_huffman(texte: str) -> dict:
    ''' Codage de Huffman optimal à partir d'un texte
    >>> codage huffman("AAAABBBBBCCD" )
    {'A': [0, 0], 'C': [0, 1, 0], 'D': [0, 1, 1], 'B': [1]}
    '''
    liste_arbres = construit_liste_arbres(texte)
    # Tri par nonbres d'occurrences décroissants
    liste_arbres.sort()
    # Tant que tous les arbres n'ont pas été fusionnés
    while len(liste_arbres) > 1:
        ## Les deux plus petits nombres d'occurrences
        # sont à la fin de la liste
        droite = liste_arbres.pop()
        gauche = liste_arbres.pop()
        new_arbre = fusionne(gauche, droite)
        # Le module bisect permet d'insérer le nouvel
        # arbre dans la liste, de manière à ce que la
        # liste reste triée
        bisect.insort(liste_arbres, new_arbre)
    # A la fin il n'en reste qu'un : notre arbre d'Huffman
    arbre_huffman = liste_arbres.pop()
    # Parcours de l'arbre pour relever les codes
    dico = {}
    parcours(arbre_huffman, [], dico)
    return dico
# Script principal
with open("texte.txt") as f:
    texte = f.read()
print (codage_huffman(texte))
```