Algorithmique
TP Algorithmes gloutons
Codage de Huffman

I. Présentation du codage de Huffman

Les codages de *Huffman* constituent une technique largement utilisée et très efficace pour la *compression de données*. En effet, des économies de 20 % à 90 % sont courantes selon les caractéristiques des données à compresser.

Remarque. Dans ce TP, les données sont considérées comme étant exclusivement textuelles (i.e des suites de caractères).

I. 1. Principe

Heuristique gloutonne.
A partir de la table contenant les fréquences d'apparition de chaque caractère d'un texte à encoder, l'algorithme glouton de Huffman détermine une manière optimale de représenter chacun de ces caractères par un mot binaire dont la longueur est inversement proportionnelle à sa fréquence d'apparition.
Autrement dit, *plus la fréquence d'apparition d'un caractère est élevée, plus la longueur du mot binaire le représentant doit être courte.*

A titre d'exemple, on considère un fichier de 100000 caractères que l'on souhaite conserver de manière compacte. On observe que les caractères du fichier apparaissent avec la fréquence donnée dans le tableau ci-dessous : seulement six caractères différents apparaissent, et le caractère `a` apparaît 45000 fois.

Caractère	a	b	c	d	e	f
Fréquence d'apparition	45000	13000	12000	16000	9000	5000
Mot binaire de longueur fixe	000	001	010	011	100	101
Mot binaire de longueur variable	0	101	100	111	1101	1100

Pour un codage de longueur fixe, on a besoin de 3 bits pour représenter chacun des six caractères. Il faut donc

$$(45000 + 13000 + 12000 + 16000 + 9000 + 5000) \times 3 = 300000 \text{ bits}$$

pour encoder entièrement le fichier.

Pour un codage de longueur variable, il ne faut plus que

$$45000 \times 1 + (13000 + 12000 + 16000) \times 3 + (9000 + 5000) \times 4 = 224000 \text{ bits}$$

pour représenter le fichier, soit une économie d'environ 25%.

I. 2. Encodage et décodage

L'encodage est toujours simple pour n'importe quel code binaire des caractères : on se contente de concaténer les mots de code binaire qui représentent les divers caractères du fichier.
Ainsi, avec le code de longueur variable précédent, on code le fichier de 3 caractères `abc` sous la forme `0101100`.

Concernant le décodage, les *codes préfixes* sont par contre souhaitables car ils simplifient grandement cette opération. En effet, comme aucun mot de code n'est un préfixe d'un autre, le mot de code binaire qui commence un fichier encodé n'est par conséquent pas ambigu. Autrement dit, il suffit d'identifier ce premier mot de code, de le traduire par le caractère initial, de le supprimer du fichier encodé, puis de répéter le processus de décodage sur le reste du fichier.
Dans notre exemple, la chaîne `001011101` ne peut être interprétée que comme `0-0-101-1101`, ce qui donne la chaine de caractères `aabe`.

II. Implémentation du codage de Huffman

L'intégralité des codes demandés est à implémenter dans le fichier `code_huffman.py`.

La première étape de la méthode de compression de données de Huffman consiste à compter le nombre d'occurrences de chaque caractère du texte à encoder.

1) Écrire une fonction `table_fréquences(texte)` qui étant donné une chaine de caractères `texte` passée en argument renvoie un dictionnaire qui associe à chaque caractère son nombre d'occurrences dans `texte`.

   A titre d'exemple, la trace en console de cette fonction pour la chaine de caractères `'ABRACADABRA'` :

   ```
   >>> table_frequences('ABRACADABRA')
   {'A': 5, 'B': 2, 'R': 2, 'C': 1, 'D': 1}
   ```

Les trois fonctions suivantes (non étudiées car hors programme de première NSI), permettent :

- `arbre_huffman(occurrences)` de construire l'arbre de Huffman à partir du dictionnaire des occurrences des caractères du texte à encoder ;
- `parcours_arbre(arbre, prefixe, codes)` et `code_huffman(arbre)` de construire, à partir du parcours de l'arbre de Huffman précédent, un dictionnaire dont les clés sont des mots binaires et les valeurs associées les caractères correspondants.

A titre d'exemple, la trace en console de ces trois fonctions pour la chaine de caractères `'ABRACADABRA'` :

```
>>> table = table_frequences('ABRACADABRA')
>>> arbre = arbre_huffman(table)
>>> codes = code_huffman(arbre)
>>> codes
{'0': 'A', '100': 'C', '101': 'D', '110': 'B', '111': 'R'}
```

2) Écrire une fonction `encodage(texte)` qui étant donné un `texte` passé en argument, renvoie la chaine binaire correspondante obtenue à partir du codage de Huffman.

Conseils méthodologiques :

- construire à partir des fonctions précédentes le dictionnaire `codes` correspondant au codage de Huffman des caractères du `texte`
- à partir du dictionnaire `codes = {mot_binaire : caractère}` construire un dictionnaire inversé `codes_inv = {caractère : mot_binaire}` ;
- parcourir de façon séquentielle le `texte` à encoder et construire la chaine binaire correspondante par concaténation des mots binaires associés à chacun des caractères.

A titre d'exemple, la trace en console de cette fonction pour la chaine de caractères `'ABRACADABRA'` :

```
>>> encodage('ABRACADABRA')
'01101110100010101101110'
```

Le taux de compression résultant de l'application du codage de Huffman, par comparaison à un encodage de type ASCII (8 bits par caractère) pris comme référence, est donné par la formule suivante :

$$\texttt{Taux} = \frac{\texttt{Nb\_bits\_Ascii} - \texttt{Nb\_bits\_Huffman}}{\texttt{Nb\_bits\_Ascii}}$$

3) Écrire une fonction `taux_compression(texte)` qui étant donné un `texte` passé en argument, calcule puis renvoie la valeur du taux de compression résultant de l'application du codage de Huffman.

A titre d'exemple, la trace en console de cette fonction pour la chaine de caractères `'ABRACADABRA'` :

```
>>> taux_compression('ABRACADABRA')
73.86
```

4) Écrire une fonction `décodage(codes, chaine_binaire)` qui étant donné un codage de Huffman `codes` et une `chaine_binaire` renvoie le texte initial.

Conseils méthodologiques :

- initialiser deux chaines de caractères vides `texte` et `tampon` ;
- parcourir de façon séquentielle la `chaine_binaire` et stocker par concaténation dans la chaîne `tampon` les valeurs binaires successives lues jusqu'à trouver une correspondance avec une clé du dictionnaire `codes` ;
- rechercher le caractère associé à cette clé puis l'ajouter par concaténation au contenu de la chaine `texte` ;
- réinitialiser la chaine `tampon`, puis réitérer le processus de décodage jusqu'à la fin de la `chaine_binaire`.

A titre d'exemple, la trace en console de cette fonction :

```
>>> codes
{'0': 'A', '100': 'C', '101': 'D', '110': 'B', '111': 'R'}
>>> decodage(codes, '011011101000101011011110')
'ABRACADABRA'
```

En conclusion, la seule connaissance de la chaine binaire issue du processus d'encodage des caractères d'un texte est-elle suffisante pour retrouver le texte initial ? Sinon, en plus de la chaine binaire, quelle(s) autre(s) informations doit-on connaitre ?