Exercice 2 (6 points)

Cet exercice porte sur les arbres binaires et la représentation binaire.

L'échange de données sur les réseaux devient de plus en plus important. Il s'avère que la compression est un élément essentiel dans cet échange. On étudie ici un algorithme de compression, reposant sur le codage de Huffman.

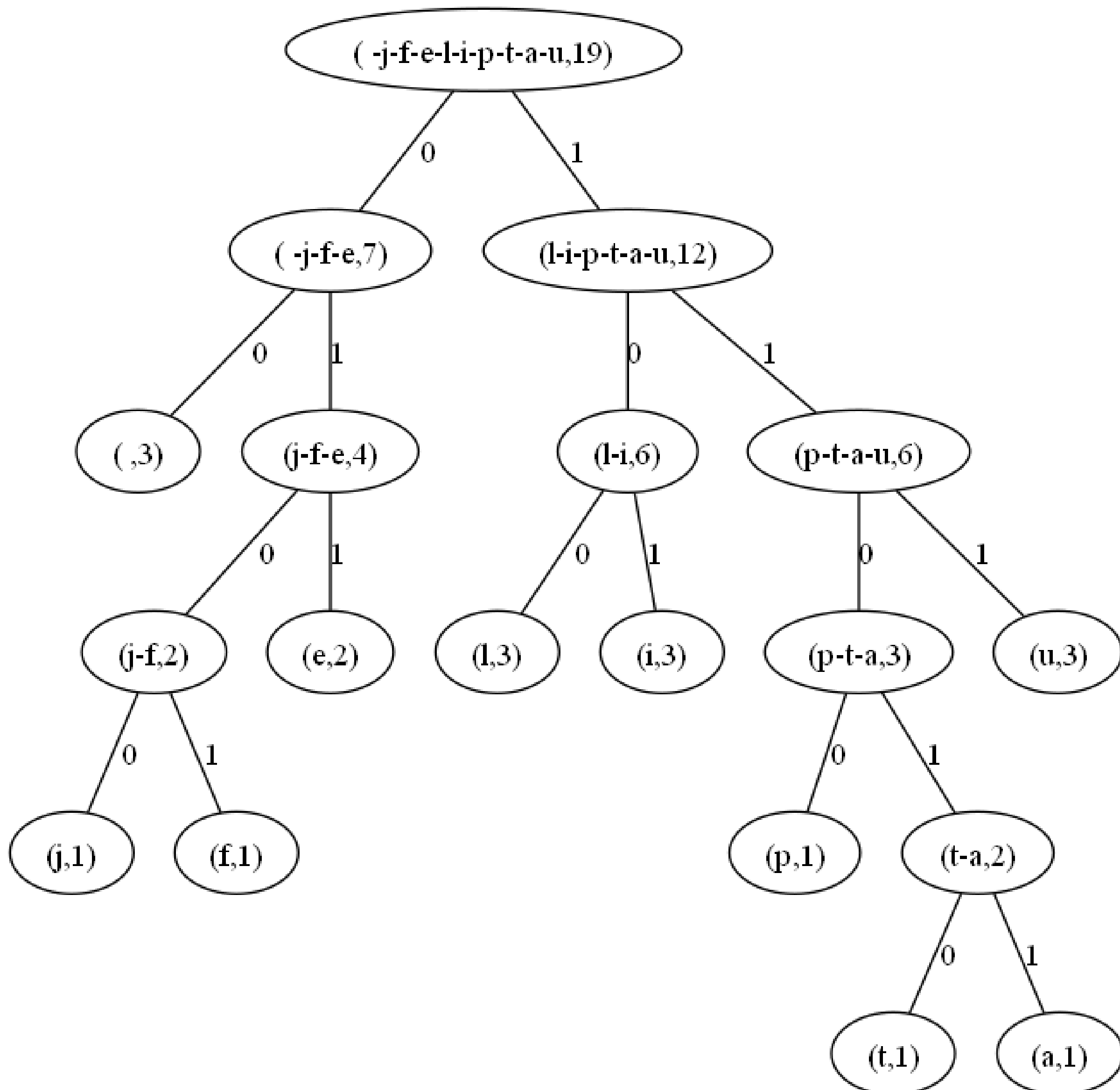

Figure 1: `arb_julie`, un arbre de compression

Le codage de Huffman est adapté au texte à compresser car on se sert du nombre d'occurrences des caractères dans le texte. Par exemple, l'arbre de la Figure 1 a été construit pour compresser la phrase `"julie fuit la pluie"`. Le nombre associé à un ensemble de caractères correspond à la somme de leur nombre d'occurrences dans la phrase. Chaque chemin parcouru de haut en bas dans l'arbre définit le code

associé au caractère présent en bas : ce code est constitué d'un `0` à chaque fois qu'on est allé à gauche et d'un `1` à chaque fois qu'on est allé à droite, mis bout à bout.

La compression de `"julie"` donne donc : `j → 0100` (gauche, droite, gauche, gauche), `u → 111`, `l → 100`, `i → 101`, `e → 011`, soit `0100111100101011`. La compression de `" "` (caractère espace) donne `00`.

1. Donner un exemple de feuille et la racine de l'arbre de la Figure 1.

On rappelle que la profondeur d'un nœud est égale à la distance en nombre d'arêtes entre ce nœud et la racine.

2. Donner la profondeur du nœud correspondant au caractère `p` et son code binaire associé.
3. On remarque que sur la Figure 1, les nœuds associés aux caractères les plus fréquents (avec un nombre d'occurrence plus élevé) ont une profondeur plus petite. Expliquer l'intérêt de cette propriété pour le codage binaire de la phrase.

On définit ci-dessous de façon incomplète la classe `Noeud`, où l'attribut `nom` est une chaîne de caractères représentant un ensemble de caractères séparés par des tirets (`-`) et où l'attribut `nb_occu` désigne la somme des nombres d'occurrences de ces caractères :

```
 1 class Noeud:
 2     def __init__(self, nom, nb_occu, fils_g, fils_d):
 3         ....nom = nom
 4         ....nb_occu = nb_occu
 5         ....fils_g = fils_g
 6         ....fils_d = fils_d
 7
 8     def __str__(self):
 9         """
10         Renvoie une chaine contenant les donnees
11         du noeud (nom et nombre d'occurrences)
12         """
13         return '(' + ... .nom + ',' + str(...) + ')'
```

4. Recopier et compléter les lignes 3, 4, 5, 6 et 13 de la classe `Noeud` ci-dessus.

On souhaite créer une fonction `liste_occurrences` qui à partir d'une chaîne de caractères `chaine` renvoie la liste des tuples `(c, nb_occu)` où `c` est un caractère apparaissant dans `chaine` et `nb_occu` est son nombre d'occurrences. Cette liste permettra par la suite de construire l'arbre qui codera chaque caractère comme en Figure 1.

Exemple :

```
>>> liste_occurrences('julie fuit la pluie')
[('j', 1), ('u', 3), ('l', 3), ('i', 3), ('e', 2),
 (' ', 3), ('f', 1), ('t', 1), ('a', 1), ('p', 1)]
```

```
 1 def liste_occurrences(chaine):
 2     dico = ...
 3     for c in chaine:
 4         if c in ...:
 5             dico[c] = dico[c] + 1
 6         else:
 7             ...
 8     liste_res = ...
 9     for cle in dico:
10         liste_res....
11     return ...
```

5. Recopier et compléter le code de la fonction `liste_occurrences` ci-dessus.

Pour élaborer l'arbre du codage de Huffman, on doit regrouper les nœuds en prenant toujours les deux nœuds de plus faible nombre d'occurrences. On va donc dans un premier temps faire un tri par insertion de la liste des tuples `(c, nb_occu)`, par ordre croissant de nombre d'occurrences.

On rappelle que la méthode `insert` de la classe `list`, utilisable avec la syntaxe : `l.insert(i, ele)`, permet d'insérer à la position `i` l'élément `ele` dans la liste `l`.

Par exemple :

```
>>> l = [0, 2, 4]
>>> l.insert(1, 8)
>>> l
[0, 8, 2, 4]
```

Le début de la fonction `tri_liste` est donné ci-dessous :

```
 1 def tri_liste(liste_a_trier):
 2     liste_triee = []
 3     for i in range(0, ...):
 4         element = liste_a_trier[i]
 5         ...
 6         while (j < len(liste_triee) and
 7                element[1] >= liste_triee[j][1]):
 8             ...
 9         liste_triee.insert(..., ...)
10     return liste_triee
```

6. Recopier et compléter le code de la fonction `tri_liste` ci-dessus.

7. Une fois la liste triée, il faut ensuite transformer la liste de tuples en liste de nœuds. Écrire la fonction `conversion_en_noeuds` qui convertit une liste de tuples en liste de nœuds.

Afin de simplifier l'élaboration de l'arbre du codage de Huffman, on doit créer une fonction qui insère un nouveau nœud dans une liste de nœuds déjà triée par ordre croissant sur le nombre d'occurrences (attribut `nb_occu`).

```
1 def insere_noeud(noeud, liste_noeud):
2     j = 0
3     while j < len(liste_noeud) and ... > ...:
4         ...
5     liste_noeud.insert(..., ...)
```

8. Recopier et compléter le code de la fonction `insere_noeud` qui modifie la liste `liste_noeud` passée en paramètre.

On va à présent créer l'arbre du codage de Huffman à partir de la liste des nœuds triée par nombre d'occurrences croissant, avec la méthode décrite ci-dessous.

Répéter les trois opérations suivantes jusqu'à ce que la liste ne soit plus composée que d'un seul nœud (la racine) :

- extraire de la liste les deux nœuds dont le nombre d'occurrences est le plus faible ;
- créer un nœud père permettant de regrouper ces deux nœuds ;
- insérer ce nœud père dans la liste.

Par exemple, la Figure 1 a été obtenue en commençant par créer le nœud `'j-f'`, les nœuds `j` et `f` ayant pour nombre d'occurences `1`.

```
1 def construit_arbre(liste):
2     while ... > 1:
3         noeud1 = liste.pop(0)
4         noeud2 = liste.pop(0)
5         nom_noeud_pere = noeud1.nom + "-" + noeud2.nom
6         nb_occu_noeud_pere = ...
7         noeud_pere = Noeud(...)
8         insere_noeud(..., liste)
9     return ...
```

On rappelle que la méthode `pop` de la classe `list`, utilisable avec la syntaxe : `l.pop(i)`, permet de retirer et renvoyer l'élément à la position `i` dans la liste `l`.

Par exemple :

```
>>> l = [0, 8, 2, 4]
>>> l.pop(1)
8
>>> l
[0, 2, 4]
```

9. Recopier et compléter le code de la fonction `construit_arbre`, qui renvoie le nœud correspondant à la racine de l'arbre.

On considère que l'on dispose d'une fonction `codage_arbre`, qui à partir d'un arbre donné en paramètre renvoie une structure telle que dans l'exemple suivant qui utilise `arb_julie`, l'arbre de la Figure 1.

Exemple :

```
>>> codage_arbre(arb_julie)
{' ': '00', 'j': '0100', 'f': '0101', 'e': '011', 'l': '100',
 'i': '101', 'p': '1100', 't': '11010', 'a': '11011',
 'u': '111'}
```

10. Indiquer la structure de données dont il s'agit.

11. Écrire une fonction `compresse` qui, à partir du texte et de la structure renvoyée par `codage_arbre`, renvoie la suite binaire sous forme d'une chaîne de caractères représentant le texte compressé.

    Par exemple on devrait obtenir :

    ```
    >>> compresse('julie', {' ': '00', 'j': '0100', 'f':
    '0101', 'e': '011', 'l': '100', 'i': '101', 'p': '1100',
    't': '11010', 'a': '11011', 'u': '111'})
    '0100111100101011'
    ```