

**Exercice 1** (6 points)

Cet exercice porte sur les arbres binaires et la programmation Python

**Partie A**

1. Avec l'arbre de codage proposé, l'espace est représenté par le mot binaire 010.
2. Le mot binaire 00.011.1010.1111.11001.1001 code le texte espion.
3. Pour obtenir les symboles par taille d'encodage croissante, on peut utiliser un parcours en largeur.

**Partie B**

4. Le calcul qui justifie le résultat de la figure 2 est  $1 + 1 + 1 + 1 + 1 + 1 + 1 + 2 + 2 = 11$  et  $3 + 4 + 4 = 11$  ce qui correspond parfaitement à l'étape 2 de l'algorithme.
5. La hauteur de l'arbre est 5, ce qui correspond au nombre maximal de bits utilisés pour coder un symbole.
6. En ASCII, la chaîne 'je pense, donc je suis' est codée sur 22 octets soit 176 bits.  
Avec Shannon-Fano et l'arbre de la figure 3, le codage est 10000001010100010010111001011010110001100110011101 sur 75 bits, et  $\frac{75}{176} \approx 0,426$  soit 42,6% donc cela permet d'utiliser environ deux fois moins d'octets.
7. On suit l'algorithme.

**étape 1** On classe les symboles du texte par nombre d'occurrences croissant.

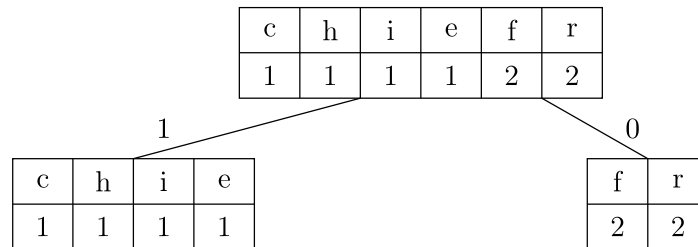
symbole	c	h	i	e	f	r
nombre d'occurrences	1	1	1	1	2	2

**étape 2** on sépare le tableau en deux de façon à avoir des effectifs aussi proches que possible dans chaque moitié, ici on a  $1 + 1 + 1 + 1 = 2 + 2$ .

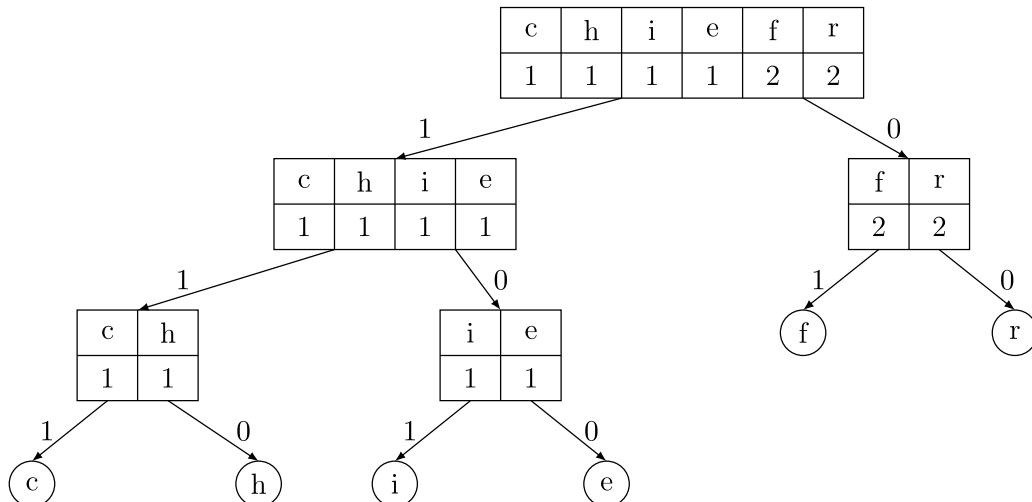
symbole	c	h	i	e
nombre d'occurrences	1	1	1	1

symbole	f	r
nombre d'occurrences	2	2

**étape 3** On commence la construction de l'arbre.



**étape 4** On recommence récursivement sur chaque sous-arbre.



**Partie C**

8. On complète les lignes 8 et 10 de la fonction `creer_dico_occ`.

```
1 def creer_dico_occ(texte):
2     """renvoie un dictionnaire dont les clés sont les
3     symboles de texte et les valeurs associées leur
4     nombre d'occurrences dans texte"""
5     dico = {}
6     for symbole in texte:
7         if symbole in dico:
8             dico[symbole] = dico[symbole] + 1
9         else:
10            dico[symbole] = 1
11    return dico
```

9. On écrit une fonction `somme_occ`.

```
1 def somme_occ(tab):
2     somme = 0
3     for symbole, nb_occ in tab:
4         somme = somme + nb_occ
5     return somme
```

10. On complète les lignes 9 et 11 de la fonction `shannon`.

```
1 def shannon(symbole, tab):
2     """renvoie l'écriture binaire associée à symbole
3     dans le tableau trié tab"""
4     if len(tab) == 1:
5         return ""
6     else:
7         t1, t2 = separe(tab)
8         if symbole in [elt[0] for elt in t1]:
9             return "1" + shannon(symbole, t1)
10        else:
11            return "0" + shannon(symbole, t2)
```

11. Les tableaux `t1` et `t2` sont strictement plus courts que le tableau `tab`, par conséquent lors des appels récursifs l'entier positif `len(tab)` est strictement décroissant et ce variant d'appel récursif assure la terminaison.

*Attention cependant, il faut pour avoir ce comportement modifier la fonction proposée, qui ne se comporte pas bien sur des exemples tels que celui-ci :*

```
>>> separe([("a", 1), ("b", 5)])
([('a', 1), ('b', 5)], [])
```

*On peut modifier les lignes 3, 4 et 5 de la fonction `separe`.*

```
1 def separe(tab):
2     moitie = somme_occ(tab) // 2
3     somme = tab[0][1]
4     i = 1
5     while i < len(tab)-1 and moitie > somme:
6         somme = somme + tab[i][1]
7         i = i + 1
8     tab1 = [tab[k] for k in range(0, i)]
9     tab2 = [tab[k] for k in range(i, len(tab))]
10    return tab1, tab2
```

12. On écrit une fonction `encode_shannon`.

```
1 def encode_shannon(str):
2     dico = creer_dico_occ(str)
3     tab = creer_tab_trie(dico)
4     code=''
5     for c in str:
6         code=code + shannon(c, tab)
7     return code
```