Métropole – 2025 – sujet2 - Correction

Exercice 1 (6 points)

Partie A – Lecture et décodage d'un arbre de codage

1. Code de l'espace

On lit les étiquettes des branches de la racine jusqu'à la feuille _ : 0 (à droite), puis 1, puis 0. Le caractère espace est codé par **010**.

2. Décodage de 000111010111111O011001

On descend dans l'arbre depuis la racine ; dès qu'on atteint une feuille, on note le symbole et on repart de la racine :

Bits	**00**	**011**	**1010**	**1111**	**11001**	**1001**
Symbole	**e**	**s**	**p**	**i**	**o**	**n**

Le texte codé est **espion**.

3. Parcours

Un **parcours en largeur** : il visite les nœuds niveau par niveau, donc les feuilles par profondeur croissante, c'est-à-dire par longueur de code croissante.

Partie B – Codage de Shannon-Fano

4. Étape 2

Le total des occurrences vaut 7 × 1 + 2 + 2 + 3 + 4 + 4 = 22. En gardant l'ordre, le groupe i, u, c, o, d, ',', p, n, j totalise 7 + 2 + 2 = **11**, et le groupe s, _, e totalise 3 + 4 + 4 = **11** : les deux sommes sont égales, c'est la séparation la plus équilibrée possible.

5. Hauteur de l'arbre

La hauteur de l'arbre de la Figure 3 est **5** (feuilles o et d). C'est la **longueur du code le plus long** (5 bits).

6. Comparaison avec le code ASCII

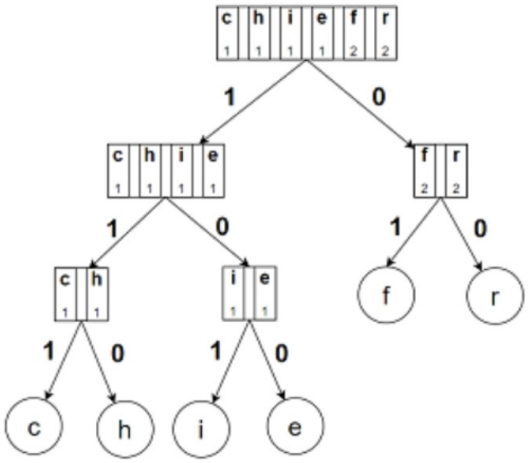
En ASCII : 22 caractères × 8 bits = **176 bits** (22 octets).

Avec Shannon-Fano, on additionne pour chaque symbole « nombre d'occurrences × longueur du code » : i, u, c, ',', p : 5 × 4 = 20 ; o, d : 2 × 5 = 10 ; n, j : 2 × 2 × 4 = 16 ; s : 3 × 3 = 9 ; _ : 4 × 3 = 12 ; e : 4 × 2 = 8, soit **75 bits**, c'est-à-dire environ 10 octets.

75 bits au lieu de 176 : on utilise bien environ **deux fois moins** de place.

7. Arbre de codage du mot « chiffrer »

Occurrences : c, h, i, e : 1 ; f, r : 2 (total 8). Première séparation : c, h, i, e (4) et f, r (4), puis on recommence :



👉*Commentaire : à occurrences égales, l'ordre des symboles est libre : d'autres arbres sont possibles. Les codes ont 3 bits pour c, h, i, e et 2 bits pour f, r.*

Partie C – Programmation Python

8. Fonction creer_dico_occ (lignes 8 et 10)

```
def creer_dico_occ(texte):
    dico = {}
    for symbole in texte:
        if symbole in dico:
            dico[symbole] = dico[symbole] + 1    # ligne 8
        else:
            dico[symbole] = 1                    # ligne 10
    return dico
```

👉*Commentaire : on parcourt le texte caractère par caractère et on met à jour le dictionnaire.*

9. Fonction somme_occ

```
def somme_occ(tab):
    s = 0
    for symbole, nb_occ in tab:
        s = s + nb_occ
    return s
```

10. Fonction shannon (lignes 9 et 11)

```
def shannon(symbole, tab):
    if len(tab) == 1:
        return ""
    else:
```

```
        t1, t2 = separe(tab)
        if symbole in [elt[0] for elt in t1]:
            return "1" + shannon(symbole, t1)    # ligne 9
        else:
            return "0" + shannon(symbole, t2)    # ligne 11
```

11. Terminaison de shannon

Le cas de base est un tableau d'un seul élément. À chaque appel récursif, tab est remplacé par t1 ou t2, qui sont **strictement plus petits** que tab (separe donne deux sous-tableaux non vides). La taille du tableau diminue donc jusqu'à 1 : la fonction termine.

12. Fonction encode_shannon

```
def encode_shannon(texte):
    tab = creer_tab_trie(creer_dico_occ(texte))
    code = ""
    for symbole in texte:
        code = code + shannon(symbole, tab)
    return code
```

👉*Commentaire : le tableau trié est calculé une seule fois ; pour chaque symbole du texte, on ajoute son code. Avec 'je pense, donc je suis', on obtient un mot de 75 bits.*