25NSIJ2ME1

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 `1000000101010001001011001011010110001100110011101` 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.

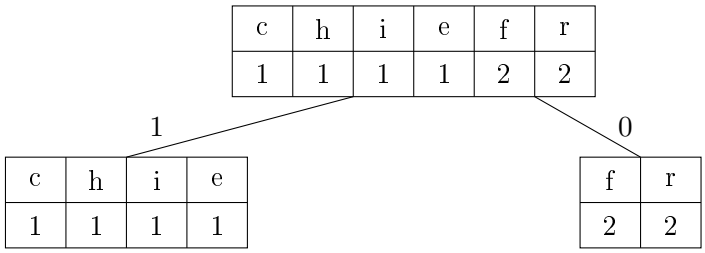
symbole	c	h	i	e	f	r
nombre d'occurrences	1	1	1	1	2	2

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

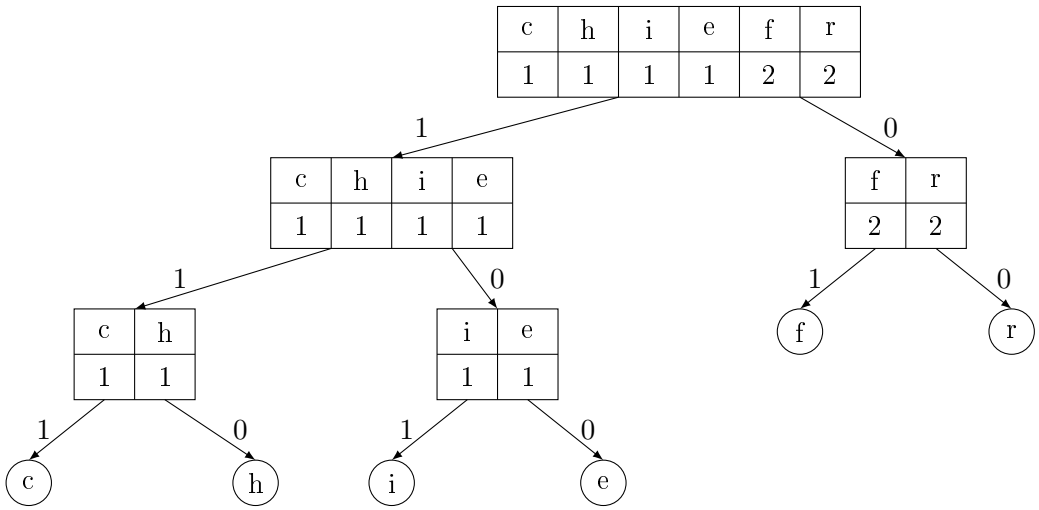
symbole	c	h	i	e
nombre d'occurrences	1	1	1	1

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`.

```
def encode_shannon(str):
    dico = creer_dico_occ(str)
    tab = creer_tab_trie(dico)
    code=''
    for c in str:
        code=code + shannon(c, tab)
    return code
```

Exercice 2 (6 points)

Cet exercice porte sur les bases de données relationnelles, le langage SQL et la programmation.

1. L'attribut `nom` n'est pas forcément unique, et ne peut donc pas être utilisé comme clé primaire de la table `adherent`.
2. La requête `SELECT nomJeu, editeur FROM jeu ORDER BY nomJeu;` renvoie la liste triée par ordre alphabétique des noms des jeux, avec l'éditeur de chaque jeu.
3. La requête `SELECT nomJeu FROM emprunt WHERE dateRendu IS NULL;` convient.
4. La requête `SELECT nom, prenom FROM adherent JOIN emprunt ON adherent.idAdherent = emprunt.idAdhere WHERE emprunt.nomJeu="Catan";` convient.
5. La requête `UPDATE emprunt SET dateRendu="2025-06-03" WHERE idEmprunt = 1538;` convient.
6. La requête `SELECT nom, categorie FROM jeu WHERE anneeSortie >= 2010 AND ageMinimum < 10;` convient.
7. On peut proposer pour la table `participation` les clés étrangères `nomEvenement` et `idAdherent` qui font respectivement référence aux tables `evenement` et `adherent`.
8. On écrit le script demandé.

```
import sqlite3

connection = sqlite3.connect("ludotheque.db")
curseur = connection.cursor()

curseur.execute("SELECT nomJeu FROM emprunt;")

jeux = curseur.fetchall()

dict_emprunts =  {}
for jeu in jeux:
    if jeu[0] in dict_emprunts:
        dict_emprunts[jeu[0]] += 1
    else:
        dict_emprunts[jeu[0]] = 1

curseur.close()
connection.close()
```

9. On écrit le script générant le podium ; on suppose qu'on a exécuté le script de la question précédente et qu'on dispose du dictionnaire `dict_emprunts`, et que les trois marches du podium sont occupées.

```
# on crée un dict d'items effectif: liste de jeux
dico = {}
for jeu in dict_emprunts:
    eff = dict_emprunts[jeu]
    if eff in dico:
        dico[eff].append(jeu)
    else:
        dico[eff]=[jeu]
```

```
    # on le transforme en liste de couples
    liste = [ (eff, dico[eff]) for eff in dico]
    # on trie cette liste dans l'ordre décroissant
    liste.sort(reverse=True)
    # on garde sur le podium les jeux des trois premiers effectifs
    podium = [liste[2][1], liste[1][1], liste[0][1]]
```

Exercice 3 (8 points)

Cet exercice porte sur la programmation de base en Python, la sécurisation des communications et les réseaux.

Partie A

1. On chiffre LIBRE avec EYQMT, on obtient PGRDX.

message	11	L	8	I	1	B	17	R	4	E
masque	4	E	24	Y	16	Q	12	M	19	T
masque+message	15		32		17		29		23	
masque+message modulo 26	15	P	6	G	17	R	3	D	23	X

2. On écrit une fonction `indice`.

```
def indice(L, element):
    for i in range(len(L)):
        if L[i] == element:
            return i
```

3. On écrit une fonction `lettres_vers_indices`.

```
def lettres_vers_indices(chaine):
    res = []
    for lettre in chaine:
        res.append(indice(alphabet, lettre))
    return res
```

ou bien

```
def lettres_vers_indices(chaine):
    return [indice(alphabet, lettre) for lettre in chaine]
```

4. On complète les lignes 7 à 13 de la fonction `chiffrement`.

```
1  def chiffrement(msg, cle):
2      assert len(cle) >= len(msg), 'impossible'
3      indices_msg = lettres_vers_indices(msg)
4      indices_cle = lettres_vers_indices(cle)
5      n = len(msg)
6      indices_msg_chiffre = []
7      for k in range(n):
8          ind = indices_msg[k] + indices_cle[k]
9          if ind >= 26:
10             ind  = ind - 26
11         indices_msg_chiffre.append(ind)
12     msg_chiffre = indices_vers_lettres(indices_msg_chiffre)
13     return msg_chiffre
```

5. Lors de l'appel chiffrement('RESEAU', 'GFTZ') on obtient une `AssertionError` à la ligne 2 de la fonction car la longueur de la clé est strictement inférieure à la longueur du message.
6. On déchiffre GMEDH avec la clé FVEIT,en complétant le tableau du bas vers le haut, on obtient .

message	1	B	18	R	0	A	21	V	14	O
masque	5	F	21	V	4	E	8	I	19	T
masque+message	6		39		4		29		33	
masque+message modulo 26	6	G	13	M	4	E	3	D	7	H

7. Pour déchiffrer en connaissant la clé, il faut soustraire modulo 26 le code de chaque lettre de la clé au code correspondant du chiffre, mais le plus simple est d'ajouter l'opposé, ce qui permet de réutiliser la fonction `chiffrement`. Il suffit de transformer A en A, B en Z, C en Y, ... et M en M pour transformer une clé de chiffrement en clé de déchiffrement, p.ex FVEIT en VFWSH.
8. On écrit la fonction `dechiffrement`.

```
def dechiffrement(msg, cle):
    assert len(cle) >= len(msg), 'impossible'
    indices_msg = lettres_vers_indices(msg)
    indices_cle = lettres_vers_indices(cle)
    n = len(msg)
    indices_msg_dechiffre = []
    for k in range(n):
        ind = indices_msg[k] - indices_cle[k]
        if ind < 0:
            ind  = ind + 26
        indices_msg_dechiffre.append(ind)
    msg_dechiffre = indices_vers_lettres(indices_msg_dechiffre)
    return msg_dechiffre
```

Partie B

9. Un algorithme de chiffrement symétrique, comme AES le masque jetable, ou utilise la même clé pour chiffrer et déchiffrer. Un algorithme de chiffrement asymétrique, comme RSA, utilise une clé publique pour chiffrer et une clé privée pour déchiffrer.
10. Bob peut déchiffrer le message envoyé par Alice grâce à sa clé privée (celle de Bob).
11. La clé publique de Bob étant publique, n'importe qui peut s'en servir pour envoyer un message chiffré à Bob. Cela assure le secret mais pas l'authentification.

    On peut aussi utiliser un protocole asymétrique pour signer un message ; Alice pourrait hasher son message, et chiffrer le hash avec sa clé privée ; alors Bob (ou n'importe qui d'autre) peut déchiffrer ce hash avec la clé publique d'Alice, ce qui authentifie son message. On peut aussi travailler sur le message entier plutôt que sur un hash, mais c'est plus coûteux en calcul.
12. Dans le protocole https, les communications sont chiffrées de bout en bout avec un protocole de chiffrement symétrique. L'établissement d'une connexion https commence par un échange de clé publiques certifiées par une autorité dont la clé publique est connue par les deux parties, ce qui permet d'authentifier ces clés. Ensuite ce protocole asymétrique sert à chiffrer et authentifier la clé secrète symétrique qui sera utilisée lors des communications.
13. Le chiffrement symétrique utilisé pendant les communications est moins coûteux en calculs que le chiffrement asymétrique, https est utilisé pour des raisons d'efficacité.

Partie C

14. Marc ne sait pas compter, il devrait changer de métier.

    Il a cherché à pinguer `192.168.100.115` alors que le poste de son collègue (qu'on espère plus compétent) est `192.168.110.115`.

    L'affichage obtenu signifie que les pong ne sont jamais arrivés chez Marc.

    `ping 192.168.110.115` est la commande adaptée.
15. Le masque 11111111.11111111.11111111.11100000 s'écrit 255.255.255.224 en décimal.
16. Le sous réseau 192.168.110.96/27 contient les adresses de 192.168.110.96 (adresse du réseau, qui se termine par 00000) à 192.168.110.127 (adresse broadcast qui se termine par 11111), soit 32 adresses dont 30 peuvent être attribuées à des interfaces, une fois qu'on a enlevé l'adresse du réseau et l'adresse de broadcast.
17. En binaire, 134 s'écrit 1000.0110.
18. Les réseaux de Bob, Marc et Zoé sont respectivement 192.168.110.96 192.168.110.128 et 192.168.110.128 donc on peut supposer que la commande utilisée était `ping 192.168.110.153` mais pour peu qu'un routeur relie les sous-réseaux n'importe laquelle des deux commandes donnait l'affichage proposé.