Rapport Codage & Cryptographie

Hammani Assia

May 6, 2024

Contents

List of Figures

1 Introduction

Ce rapport présente les etapes realisées afin de dechiffrer un message chiffré retrouvé dans les archives de Claude Shannon, un pionnier de la théorie de l'information. Le message, codé en binaire, présente des certaines caractéristiques qui nécessitent des etapes particulieres afin de pouvoir lire et comprendre le message present.
pour atteindre cette objectif, nous analyserons la lettre code en binaire pour detecter les eventuels erreurs et pouvoir les corriger, puis nous convertirons la lettre en caractere ASCII pour pouvoir ensuite la dechiffrer grace a une méthode de chiffrement par substitution polyalphabétique. Enfin nous pourrons la chiffrer et la conpresser pour pouvoir l'envoyer au destinataire.
L'objectif est d'utiliser les notions vu en cours afin de pouvoir envoyer la lettre a son destinataire a la fin.

2 Comprehension du sujet

Tous d'abord les etapes a réalise sont d'examiner la lettre.txt en detectant les erreurs puis les corriger. Pour cela nous utilisons le code de Hamming pour détecter et corriger les erreurs. En effet, le code de Hamming est une technique de codage d'erreur qui permet de détecter, mais aussi de corriger les erreurs qui peuvent se produire lors de la transmission de données.

3 Detection et correction des erreurs

L'objectif de cette partie est de montrer et expliquer les etapes realisé afin de detecter les erreurs dans la lettre et de les corriger.

3.1 Detection d'erreur

Tout d'abord nous savons que la lettre est codé en binaire avec des 0 et 1 et qu'il y a 47 712 bits dans la lettre. Nous savons que nous devons realiser le code de hamming d'apres le sujet et nous avons donc decider de tester l'hypothese du code de hamming(7,4) qui a travers un message de sept bits transfère quatre bits de données et trois bits de parité. Nous décidons donc

de vérifier cette hypothèse. Pour cela, nous effectuons les calculs suivants:

$$\frac{47712}{7} = 6816$$

Si on supprime les 3 bits de parité dans chaque séquence de 7 bits, nous aurons 4 bits restants dans chaque séquence. Donc après la suppression dans les 6816 séquences, nous aurons:

$$\text{Nombre de bits total} = 6816 \times 4 = 27264$$

Ainsi,

$$27264 \times 1.75 = 47712$$

Nous remarquons que l'hypothese est bien correcte. Nous savons que pour chaque sequence de 7 bits nous avons bien 4 bits d'infos et 3 bits de parite. Pout detecter les erreurs nous avons utiliser une fonction `calculer_parite(mot, positions)` qui calcule le bit de parité pour un mot donné en fonction des positions spécifiées.
En effet calculer la parite permet de comparer et verifier l'exactitude des bits de parité calculés des bits de parité fournis. Si les deux ensembles de bits de parité sont identiques, alors la parité est correcte sinon cela signifie qu'il y a probablement eu une erreur dans la transmission. Le code de Hamming (7,4) utilise trois bits de parité pour chaque mot de 4 bits. Chaque bit de parité est calculé en fonction de différentes combinaisons de bits dans le mot de 4 bits. les combinaisons sont les suivantes:

$$c_1 = \begin{cases} 0 & \text{si } a_1 + a_2 + a_3 \text{ est pair,} \\ 1 & \text{sinon,} \end{cases}$$

$$c_2 = \begin{cases} 0 & \text{si } a_1 + a_2 + a_4 \text{ est pair,} \\ 1 & \text{sinon,} \end{cases}$$

$$c_3 = \begin{cases} 0 & \text{si } a_2 + a_3 + a_4 \text{ est pair,} \\ 1 & \text{sinon.} \end{cases}$$

Une sequence equivaut $a_1a_2a_3a_4c_1c_2c_3$ avec $a_1a_2a_3a_4$ le mot d'infos et $c_1c_2c_3$ les bits de parité.

Par exemple, `calculer_parite(mot, positions)` 1 parcourt chaque position dans une liste positions et pour chaque position, elle décale les bits du mot vers la droite de i positions, puis effectue une opération ET bit à bit avec 1. On aura ainsi le bit à la position i du mot. Elle ajoute ensuite le bit extrait à la variable 'parite', parcourt toutes les positions puis retourne la parite % 2.
Ainsi si le nombre de bits à 1 est pair la parite est 0, sinon 1.

De cette façon, si un bit est incorrect, il y aura une incohérence dans deux bits de parité, ce qui permettra de localiser l'erreur d'apres les regles suivantes:

- $a1$ est erroné $\leftrightarrow$ $c1$ et $c2$ erronés, $c3$ correct
- $a2$ est erroné $\leftrightarrow$ $c1$, $c2$ et $c3$ erronés
- $a3$ est erroné $\leftrightarrow$ $c1$ et $c3$ erronés, $c2$ correct
- $a4$ est erroné $\leftrightarrow$ $c2$ et $c3$ erronés, $c1$ correct

Nous avons egalement la fonction `verifier_mot_avec_parite(mot_avec_parite)` 1 qui prend le mot de 7 bits donc le mot d'infos et les bits de parite, supprime les bits de parite en realisant un decalage des bits vers la droite et une opération ET bit à bit avec 1. Puis elle appelle la fonction `verifier_mot(mot, c)` avec le mot d'info.

Nous avons ensuite la fonction `verifier_mot(mot, c)` 1 utilisé dans la fonction précedente qui vérifie les bits de parité d'un mot donné et elle renvoie un nombre indiquant quels bits de parité est incorrect. Pour cela on définit une liste de positions pour chaque bit de parité. Ici nous avons notre mot d'infos a 4 bits: $a_1a_2a_3a_4$ avec a1 pour la position 3, a2 pour la position 2, a3 pour 1 et a4 pour 0. Elle vérifie chaque bit de parité en calculant le bit de parité du mot et en le comparant au bit de parité attendu.Si un bit ne correspond pas au bit de parité attendu, elle renvoie l'erreur.

3.2 Correction d'erreur

Apres avoir trouvé la position des erreurs il nous faut maintenant les corriger. Pour cela nous avons une fonction `corriger_erreur(mot_avec_parite, erreurs)` qui permet de corriger une erreur dans le mot.

Cette fonction prend en parametre un mot avec sa parite et une valeur d'erreur qui est obtenue a partir de la fonction `verifier_mot(mot, c)` vue

précedemment. Dans la fonction `corriger_erreur(mot_avec_parite, erreurs)` nous avons des conditions if qui permettent de determiner l'index du bit erroné dans le mot avec parité. Une fois cette index identifié, la valeur est inverser pour corriger l'erreur. Cette fonction se termine en renvoyant le mot avec parite corrigé.

De plus nous utilisons une fonction `copier_fichier(nom_fichier_source, nom_fichier_destination)` pour copier le contenu de la lettre.txt avec les erreurs corrigés et les bits de parité supprimé. En effet une fois les erreurs corrigés dans les mots d'informations nous supprimons les bits de parite qui ne sont plus utile vers un fichier de destination.
Pour cela on ouvre un fichier source en mode lecture binaire avec rb et un fichier de destination en mode écriture avec w. En effet, le mode rb indique que le fichier est traité comme une séquence d'octets bruts et le mode w ouvre un fichier en écriture. Cela nous permettra d'ecrire le resultat de notre lettre.txt corrigé et les bits de parité supprimé dans ce fichier d'ecriture.
La fonction lit ensuite le fichier source bit par bit. Pour chaque bit, elle décale `mot_avec_parite` d'un bit vers la gauche et ajoute le bit lu. Lorsque 7 bits sont lu, elle vérifie si le mot avec parité comporte des erreurs en utilisant la fonction `verifier_mot_avec_parite(mot_avec_parite)`. Si des erreurs sont détectées, elle corrige ces erreurs en utilisant la fonction `corriger_erreur(mot_avec_parite, erreurs)`.
Apres cela nous supprimons les bits de parité pour ne garder que les bits d'information. Pour cela, on utilise l'opération AND avec le masque 0b01111000. Cette opération permet de mettre à zéro les bits qui ne sont pas des bits d'information c'est a dire les bits de parité. Ensuite on décale les bits restants vers la droite pour qu'ils soient à la bonne position.
Par exemple si nous avons le mot: 0001011 nous appliquons l'operation suivante:

$$\begin{array}{r} 0\,0\,0\,1\,0\,11 \\ \&\ 0\,1\,1\,1\,0\,00 \\ \hline 0\,0\,0\,0\,0\,00 \end{array}$$

Puis nous decalons le résultat de 3 bits vers la droite pour supprimer les bits de parité via l'opération:
$0000000 >> 3 = 0000000$

Enfin nous avons le mot sans parité est 0000.
Nous faisons cette operation pour tous les mots de la lettre.txt et affichons le resultat dans le fichier lettreResultatTEST.txt.

4 Conversion en ASCII

Nous devons a présent convertir la lettre resultat code en binaire en caractere ASCII. Pour cela nous avons une fonction `copier_fichier_convertir_ascii(nom_fichier_source, nom_fichier_destination)` qui ouvre le fichier source donc ici, lettreResultatTEST.txt en mode lecture (r) et le fichier de destination en mode écriture (w). Cette fonction lit ensuite le fichier source 8 bits par 8 bits. Pour chaque séquence lu, elle convertit les bits en un entier en base 2 en utilisant la fonction int(bits, 2), puis convertit l'entier en un caractère ASCII grâce a la fonction chr(entier). la fonction réalise ces actions pour chaque séquence dans tous le fichier source et une fois tous les bits lu et converti, elle ecrit le resultat dans le fichier de destination.
Le resultat en ASCII se trouve dans le fichier lettreResultatASCII.txt.

5 Dechiffrement

Apres avoir traduit la lettre en ASCII nous nous retrouvons avec une suite de caractère qui n'est pas compréhensible. Il nous faut maintenant dechiffrer cette lettre. Pour déchiffrer cette lettre qui a été chiffré, nous avons besoin de connaître l'algorithme de chiffrement utilisé ainsi que la clé de chiffrement. Pour transformer la lettre chiffré en texte décodé nous devons utiliser la même méthode et la même clé qui ont été utilisées pour le codage, mais appliqué de manière inverse. Nous savons d'après le sujet que la lettre a été chiffrée par une méthode de chiffrement polyalphabétique du XVIème siècle. Après quelques recherches nous avons trouve que cela correspondait au chiffrement de Vigenère. A propos du chiffrement de Vigenere, nous savons qu'il est basé sur les 26 lettres de l'alphabet latin. La clef secrète est une chaîne de caractères de longueur secrète. Concernant la cle dans le sujet il est dit "Grand amateur de serpents, il parvient à la décrypter" donc nous supposons que la cle a un rapport avec les serpents. Nous tentons un rapporchement, nous sommes en informatique et le mot a trouve a un rapport avec les serpents nous pensons biensur au langage de programmation Python. Les hypotheses a tester sont donc:python;Python. De plus nous savons que le chiffrement de Vigenère est conçu pour fonctionner avec les lettres de l'alphabet, et non avec d'autres types de caractères. Cependant nous ne savons pas si la lettre contenait des ponctuations.. Nous devons donc nous concentrer sur l'alphabet seulement. Concernant la langue utilisé dans la lettre nous

prenons le debut de la lettre:"Ac 02 fhff 1952" nous comprenons très vite que la lettre commence par la date enoncé. Les langues les plus probables sont le francais et l'anglais. Cependant nous savons que en français, la date du jour s'écrit : jour mois année alors qu'en anglais elle s'ecrit mois jour, année. Dans notre cas nous avons un article puis un nombre, un mois a 4 lettres et enfin un nombre. Nous en deduisons donc que la lettre est ecrite en francais.

En premier lieu nous decidons de ne pas prendre en compte les caracteres spéciaux tel que les ponctuations,ect. Pour dechiffrer nous avons utilisé uniquement l'alphabet de base (sans accents) pour le déchiffrement. Cela garantit que si le texte original ne contient pas de caractères accentués, le texte déchiffré n'en contiendra pas non plus.

Concernant la fonction de dechiffrement, nous utilisons deux variables pour gérer les chaînes de caractères, une pour les lettres majuscules et une pour les lettres minuscules. La variable 'cle' est définie et convertie en majuscules pour correspondre à l'alphabet en majuscules. Ensuite, nous utilisons une boucle 'for' pour parcourir les lettres chiffrées. À l'intérieur de cette boucle, des conditions 'if' sont utilisées pour vérifier si le caractère lu est alphabétique, en majuscules, en minuscules ou non alphabétique, et nous effectuons des actions en conséquence. Nous calculons l'index en soustrayant l'index du caractère actuel de la clé dans l'alphabet de l'index du caractère chiffré dans l'alphabet. Si nous dépassons la longueur de la clé, nous revenons au début. La fonction nous renvoie le texte déchiffré. Nous avons reussi a dechiffrer la lettre et son contenu est dans le fichier lettreResultatVigenere.txt.

6 Chiffrement

Apres avoir dechiffre la lettre et decouvert son destinataire, il nous faut maintenant la chiffrer. Dans le sujet il nous est dit " il rechiffre la lettre par une variante du chiffrement précédant, considéré comme "le seul algorithme cryptographique à confidentialité parfaite" par Claude Shannon." Nous en deduisons que nous devons chiffrer le fichier lettreResultatVigenere.txt par une variante du chiffrement de vigenere. On nous donne un indice pour trouver cette variante, il est dit qu'il est considére comme le seul algorithme a confidentialité parfaite par claude Shannon. Un chiffre parfaitement sûr est un chiffre tel que, l'adversaire interceptant le message, même ayant à sa

disposition une puissance de calcul infinie, ne peut pas retrouver la moindre information concernant le message clair à partir du message chiffré. Apres quelques recherches nous trouvons que cette variante est le chiffre de Vernam. En effet, Claude Shannon a prouvé en 1949 que ce chiffre était parfaitement sûr. La seule information dont dispose une personne qui a pu intercepter le message chiffré est la longueur du message clair. De plus, tout chiffre parfaitement sûr est nécessairement une variante du chiffre de Vernam. Donc nous avons trouvé la variante, il nous faut maintenant convertir la lettre en suite de bit de 0 et 1 et nous prenons une clé (complètement aléatoire) composée elle aussi d'une suite de 0 et de 1, aussi longue que le message à chiffrer. On prend ensuite chaque bit du message clair et de la clé, et on en fait le ou exclusif. Cependant ce chiffrement exige qu'une clé serve une seule fois. Si nous utilisons la même clé deux fois, alors on peut extraire beaucoup d'informations des messages chiffrés. Nous avons donc utilise une fonction `string_to_bits(s)` pour convertir la lettre ascii en binaire.

Pour genere une cle aleatoire nous avons utilisé une fonction `vernam_cipher(text, key)` . Dans cette fonction nous avons les expressions text[i:i+8] et key[i:i+8] qui prennent des sous-chaînes de 8 bits à la fois de la lettre et de la clé. Puis on convertit les sous chaine en entier en base 2. Apres avoir convertit en entier nous effectuons une operation XOR bit a bit. Nous avons donc le fichier lettreResultatVernam.txt qui contient la lettre chiffrée et le fichier CleVernam.txt qui contient la cle.

7 Compression

Dans le sujet on nous parle de Claude Shannon, Richard Hamming et David Albert Huffman. Nous avons utilise le code de Hamming pour detecter et corriger les erreurs, puis nous avons utilise l'algorithme cryptographique à confidentialité parfaite de Claude Shannon pour chiffrer la lettre et sa cle. A présent il nous reste a compresser la lettre et sa cle et pour cela nous avons l'algorithme de compression Huffman qui est un algorithme de compression sans perte qui utilise la théorie de l'information pour réduire la taille des données.
Pour cela , l'algorithme compte le nombre d'occurence de chaque caractere dans la lettre. Grâce a ces nombres d'occurences, l'algorithme construit un arbre binaire dans lequel les symboles les plus fréquents sont placés près de la racine, tandis que les autres sont placés plus loin. Ensuite on utilise le code

binaire pour codés les caracteres. Les caracteres avec le nombre d'occurences le plus élevé auront des codes plus court que les autres. Dans notre cas nous avons une lettre codé en binaire nous allons donc commencer par compter le nombre d'occurence de chaque octet.
Nous avons rencontré des difficultés sur le fait de compter le nombre d'occurence dans le fichier binaire mais une fois ce probleme résolu nous avions pas le bonne arbre,ect.. Après plusieurs erreurs nous avons décider de réaliser la compression sur la lettre dechiffrée en caractere ascii. Pour cela nous avons utilisé la fonction count_occurrences(text) qui compte le nombre d'occurrences de chaque caractère pour chaque sequence de 8 bits. Les valeurs sont placés dans un tuple avec la valeur de l'occurence et le caractère correspondant. Nous avons utilisé la fonction ord pour convertir le caractère en code ASCII, puis nous utilisons ce code comme index pour accéder à l'élément correspondant dans la liste letters et incrémenter le compteur d'occurrences.
Nous avons ensuite une fonction build_tree(letters) qui construit l'arbre de Huffman. Dans cette fonction on parcourt chaque element de 'letters' et on les ajoute dans la liste nodes sous forme de tuple. Nous avons une boucle qui tant qu'il reste 2 noeuds, va combiner en paire les feuilles avec les frequences les plus faibles jusqu'a qu'il ne reste que le noeud racine. Nous avons egalement la fonction create_dict(tree) qui crée un dictionnaire en fonction des fréquences des caractères. Nous spécifions que la longueur du code binaire est égale à la profondeur dans l'arbre et pour chaque lettre, nous partons de la racine et on ajoute un 0 si nous prenons la branche de gauche et un 1 pour la branche de droite. Enfin, nous avons la fonction compress_file(input_file) qui permet d'ouvrir le fichier que l'on souhaite conpresser, d'éxécuter les fonctions précédentes et d'afficher la taille du fichier avant et après compression.

8 Conclusion

Ce projet nous a permis de revoir les notions abordes en cours tels que la detection et correction d'erreur, le chiffrement, déchiffrement.

9 Annexes

Ci dessous le resultat avec affichage des bits erronés:
Ci dessous les fonction de detection d'erreurs:

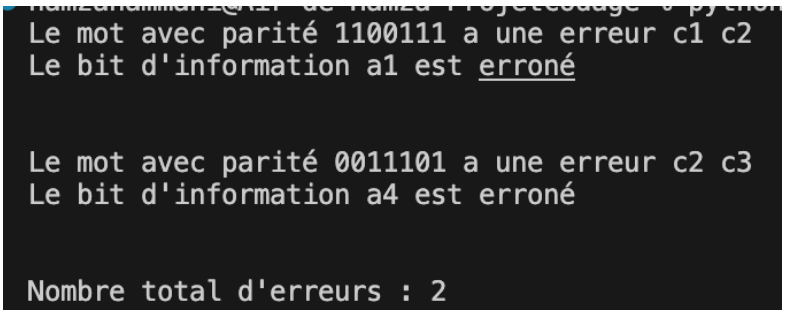

Figure 1: Affichage des positions d'erreurs

```python
def calculer_parite(mot, positions):
    parite = 0
    for i in positions:
        parite += (mot >> i) & 1
    return parite % 2

def verifier_mot(mot, c):
    erreurs = 0
    positions = [[3, 2, 1], [3, 2, 0], [2, 1, 0]]
    for i in range(3):
        if calculer_parite(mot, positions[i]) != c[i]:
            erreurs |= 1 << i
    return erreurs

def verifier_mot_avec_parite(mot_avec_parite):
    mot = mot_avec_parite >> 3
    c = [(mot_avec_parite >> 2) & 1, (mot_avec_parite >> 1) & 1,
    ↪  mot_avec_parite & 1]
    return verifier_mot(mot, c)
```

Ci dessous la fonction de correction d'erreurs:

```python
def corriger_erreur(mot_avec_parite, erreurs):
```

```
    # Trouver l'index du bit erroné
    index_bit_erreur = 0
    if ((erreurs & (1 << 0)) and (erreurs & (1 << 1)) and not
    ↪  (erreurs & (1 << 2))):
        index_bit_erreur = 6  # Le bit d'information a1 est erroné
    elif ((erreurs & (1 << 0)) and (erreurs & (1 << 1)) and
    ↪  (erreurs & (1 << 2))):
        index_bit_erreur = 5  # Le bit d'information a2 est erroné
    elif ((erreurs & (1 << 0)) and not (erreurs & (1 << 1)) and
    ↪  (erreurs & (1 << 2))):
        index_bit_erreur = 4  # Le bit d'information a3 est erroné
    elif (not (erreurs & (1 << 0)) and (erreurs & (1 << 1)) and
    ↪  (erreurs & (1 << 2))):
        index_bit_erreur = 3  # Le bit d'information a4 est erroné

    # Inverser le bit erroné
    mot_avec_parite ^= (1 << index_bit_erreur)

    return mot_avec_parite
```

Ci dessous la fonction de copie du resultat apres correction et suppression des bits de parité:

```
def copier_fichier(nom_fichier_source, nom_fichier_destination):
    with open(nom_fichier_source, "rb") as fichier_source,
    ↪  open(nom_fichier_destination, "w") as fichier_destination:
        mot_avec_parite = 0
        nombre_bits = 0

        while (bit := fichier_source.read(1)) != b'':
            mot_avec_parite = (mot_avec_parite << 1) | (bit[0] & 1)
            nombre_bits += 1

```

```
            if nombre_bits == 7:
                erreurs = verifier_mot_avec_parite(mot_avec_parite)
                if erreurs:
                    mot_avec_parite =
                    ↪ corriger_erreur(mot_avec_parite, erreurs)
                # Supprimer les bits de parité
                mot_sans_parite = (mot_avec_parite & 0b0111000) >>
                ↪ 3
                # Écrire le mot sans les bits de parité en binaire
                ↪ dans le fichier de destination
                fichier_destination.write(f"{mot_sans_parite:04b}")

                mot_avec_parite = 0
                nombre_bits = 0

    print("Copie du fichier terminée avec succès.")
```

Ci dessous la fonction de copie du resultat apres conversion en ASCII:

```
def copier_fichier_convertir_ascii(nom_fichier_source,
↪ nom_fichier_destination):
    with open(nom_fichier_source, "r") as fichier_source,
    ↪ open(nom_fichier_destination, "w") as fichier_destination:
        ascii_data = ""
        while (bits := fichier_source.read(8)) != '':
            # Convertir les bits en un entier en base 2
            entier = int(bits, 2)
            # Convertir l'entier en caractère ASCII
            caractere_ascii = chr(entier)
            ascii_data += caractere_ascii
        fichier_destination.write(ascii_data)
    print("Conversion du fichier en ASCII terminée avec succès.")
```

Ci dessous la fonction de déchiffrement de Vigenère:

code:vigenere

```python
def dechiffrement_vigenere(texte_chiffre, cle):
    alphabet_maj = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    alphabet_min = 'abcdefghijklmnopqrstuvwxyz'
    cle = cle.upper()
    texte_clair = ''
    index_cle = 0
    for i in range(len(texte_chiffre)):
        if texte_chiffre[i].isalpha():  # permet d'ignorer les
        ↪  caractères non alphabétiques
            if texte_chiffre[i].isupper():
                alphabet = alphabet_maj
                if texte_chiffre[i] not in alphabet:  # si
                ↪  caractère n'est pas dans l'alphabet,on l'ignore
                    continue
                index_chiffre = alphabet.find(texte_chiffre[i])
                index_cle_actuel = alphabet.find(cle[index_cle %
                ↪  len(cle)])
            else:
                alphabet = alphabet_min
                if texte_chiffre[i] not in alphabet:
                    continue
                index_chiffre = alphabet.find(texte_chiffre[i])
                index_cle_actuel = alphabet.find(cle[index_cle %
                ↪  len(cle)].lower())
            # Calculer l'index de la lettre
            index_texte = (index_chiffre - index_cle_actuel) %
            ↪  len(alphabet)
            texte_clair += alphabet[index_texte]
            index_cle += 1
        else:
            texte_clair += texte_chiffre[i]  # Ajouter les
            ↪  caractères non alphabétiques sans les changer
```

```
27      return texte_clair
```

Ci dessous la fonction de conversion de ASCII a binaire:

code:conversionBinaire

```
1       def string_to_bits(s):
2       result = ''.join(format(c, '08b') for c in bytearray(s,
        ↪  "utf-8"))
3       return result
```

Ci dessous la fonction de generation de cle aleatoire:

code:cle

```
1   def generate_binary_key(n):
2
3       binary_string = ''.join(random.choice('01') for _ in range(n))
4
5       return binary_string
6
```

Ci dessous la fonction de generation de la fonction du chiffre de Vernam:

code:cle

```
1   def vernam_cipher(text, key):
2       return "".join(format(int(text[i:i+8], 2) ^ int(key[i:i+8], 2),
        ↪  '08b') for i in range(0, len(text), 8))
3
```

Ci dessous la fonction qui génère le nombre d'occurences pour chaque valeur:

code:cle

```
1   def count_occurrences(text):
2       letters = [[0, chr(i)] for i in range(256)]
3       for char in text:
4           letters[ord(char)][0] += 1
5       return letters
```

Ci dessous la fonction qui génère l'arbre de Huffman:

code:cle

```
def build_tree(letters):
    # parcourt chaque element de la liste letters et les ajoute dans la liste nodes sous forme de tuple
    nodes = [(k, v) for (k, v) in letters]
    # on prend les deux noeuds (ou feuilles) de frequence le plus faible on en fait un noeud, de poids la somme des deux petits poids
    # on boucle jusqu'a qu'il reste au moins 2 noeuds
    while len(nodes) >= 2:
        small_min = (0, nodes[0]) # noeud des frequence min
        big_min = (1, nodes[1])
        for i in range(2, len(nodes)):
            if nodes[i][0] <= small_min[1][0]:  #si le poids du noeud i est inferieur au poids du noeud small_min alors on echange les deux
                big_min = small_min
                small_min = (i, nodes[i])
            elif nodes[i][0] <= big_min[1][0]:  # sinon si le poids du noeud i est inferieur au poids du noeud big_min alors on echange les deux
                big_min = (i, nodes[i])
        new_node = ( # on cree un nouveau noeud avec les deux noeuds de frequence min
            small_min[1][0] + big_min[1][0],
            nodes[small_min[0]],
            nodes[big_min[0]]
        )
        #  on ajoute le nouveau noeud
        nodes[small_min[0]] = new_node
        nodes.pop(big_min[0])

```

```
    #si il reste un seul noeud alors on le renvoie
    return nodes[0]
```

Ci dessous la fonction qui génère le dictionnaire avec les codages binaires:

code:cle

```
def create_dict(tree):
    dictionnaire = [("", tree)]
    dictionary = {}
    while dictionnaire:
        code, node = dictionnaire.pop(0)  # on prend le premier element de la liste dictionnaire et on le supprime
        if len(node) == 2:  # si c'est une feuille alors on ajoute la lettre et son code dans le dictionnaire
            dictionary[node[1]] = code
        elif len(node) == 3:  # sinon on ajoute les deux fils du noeud dans la liste dictionnaire
            # Gauche -> 0, droite -> 1
            dictionnaire.append((code + "0", node[1]))# on ajoute le fils gauche du noeud dans la liste dictionnaire
            dictionnaire.append((code + "1", node[2]))
    return dictionary
```

Ci dessous la fonction de compression:

code:cle

```
def compress_file(input_file):
    with open(input_file, 'r') as f:
        text = f.read()

    occurrences = count_occurrences(text)
    tree = build_tree(occurrences)
    dictionary = create_dict(tree)
    compressed_text = "".join(dictionary[char] for char in text)

```

```
    return compressed_text, dictionary


input_file = "lettreResultatVigenere.txt" #le fichier a compresser

compressed_text, dictionary = compress_file(input_file)

with open(input_file, 'r') as f:
    text = f.read()

print("Nombre de bits avant compression: {} bits / Nombre de bits
↪  après compression : {} bits".format(len(text) * 8,
↪  len(compressed_text)))
```