DS02

Algorithmique et programmation

Sources : exercice 1 : Clément Roux -UPSTI exercice 2 : Clément Roux -UPSTI

Proposition de corrigé

Exercice 1 : Décryptage de texte

Q 1 :

```
mess="la metamorphose"
alphabet="abcdefghijklmnopqrstuvwxyz"
code=""
```

Q 2 :

Avec un décalage de $n = 3$, "franz" est codé "iudqc"

Q 3 :

```
def decalage(c,n):
    ind=alphabet.index(c)
    return alphabet[(ind+n)%26]
```

Q 4 :

```
def chiffrement_cesar(mess,n):
    code=""
    for i in range(len(mess)):
        if mess[i].isalpha():
            code=code+decalage(mess[i],n)
        else:
            code=code+mess[i]
    return code
```

Q 5 :

```
def decryptage_cesar(code):
    for n in range(len(alphabet)):
        print(chiffrement_cesar(code,-n)," avec n= ",n)
    return None
```

Q 6 :

Ce code ne possède que 26 clés différentes. On sait que dans l'alphabet Français les lettres ne possèdent pas les mêmes fréquences d'apparition. Par exemple le 'e' est beaucoup plus répandu que le 'y'. Paradoxalement plus le code serait long et plus il serait simple à décrypter en trouvant les caractères les plus répandus et en supposant qu'ils correspondent au "e". On peut aussi analyser les caractère présent à côté des apostrophes et supposés qu'il doivent correspondre au "m" ou au "l".

Q 7 :

Ces instruction permettent d'afficher 26 lignes.

Les deux première lignes affichées seront les suivantes :

bcdefghijklmnopqrstuvwxyza

cdefghijklmnopqrstuvwxyzab

Q 8 :

```
def generer_table():
    alphabet= "abcdefghijklmnopqrstuvwxyz"
    table = []
    for i in range(26):
        ligne=[]
        ligne = [c for c in alphabet]
        table.append(ligne)
        alphabet = alphabet[1:]+alphabet[:1]
    return table
```

Q 9 :

La première méthode utilise la concaténation un très grand nombre de fois. Cette méthode n'est pas forcément très efficace. Alors que la deuxième méthode permet de limiter le nombre de concaténation en déterminant le nombre de concaténation à l'avance par le reste de la division euclidienne.

Q 10 :

```
def code_vigenere(mot,cle):
    alphabet= "abcdefghijklmnopqrstuvwxyz"

    # On genere la table de vigenere
    table = generer_table()
    # On genere la clef
    tab_cle = generer_cle_1(mot,cle)
    # On genere le code
    code = ""
    for i in range(len(mot)):
        ligne = mot[i]
        col   = tab_cle[i]
        id_ligne = alphabet.index(ligne)
        id_col = alphabet.index(col)
        code = code+table[id_ligne][id_col]
    return code
```

Q 11 :

Le chiffre de Vigenère est une amélioration de la méthode de César, son principal intérêt réside dans l'utilisation non pas d'un, mais de 26 alphabets décalés pour chiffrer un message que l'on retrouve dans le carré de Vigenère (d'où l'appellation polyalphabétique). Il est ainsi bien plus difficile à casser que celui de César, on passe d'une clé sous la forme d'un nombre entier de $d \in [1,25]$ à une clé sous la forme d'une chaine de caractère de longueur inconnue. Toutefois 300 ans après sa création plusieurs techniques permettant de casser cette méthode de chiffrement ont été développées.

Exercice 2 : Les carrés magiques

Q 12 :

15	22	9	16	3
2	14	21	8	20
19	1	13	25	7
6	18	5	12	24
23	10	17	4	11

On vérifie les trois propriétés d'un carré magique avec La somme de chaque colonne, la somme de chaque et la somme de chaque diagonale est égale à la densité $d = 65$.

Q 13 :

```
def Carre_vide(n):
    if n%2==0:
        print("Erreur n doit être impair")
        return []
    else:
        carre=[]
        for i in range(n):
            carre.append([0]*n)
        return carre
```

Q 14 :

```
def Remplir_carre(carre):
    n=len(carre)
    carre_magique=deepcopy(carre)
    x,y=(n-1)//2-1,(n-1)//2
    for i in range(1,n**2+1):
        carre_magique[y][x]=i
        print(carre_magique)
        if i%(n)==0:
            x=(x-2)%n
        else:
            x,y=(x-1)%n,(y-1)%n
    return carre_magique
```

Q 15 :

On vérifie les trois propriétés d'un carré magique avec les sommes suivantes égales à la densité :

- La somme de chaque colonne
- La somme de chaque ligne
- La somme de chaque diagonale

```
def Verif_carre(carre_magique):
    n=len(carre)
    dens=int(n*(n**2 + 1)/2)
    for i in range(n):
        d1=0
        d2=0
        for j in range(n):
            d1+=carre_magique[j][i]
            d2+=carre_magique[i][j]
        if d1!=dens or d2!=dens:
            return False
    d1=0
    d2=0
    for i in range(n):
        d1+=carre_magique[i][i]
        d2+=carre_magique[n-1-i][i]
    if d1!=dens or d2!=dens:
        return False
    return True
```