TD utilisation de dictionnaires

Exercice 1

1. Écrire une fonction `occurences(L: list) -> dict` qui renvoie un dictionnaire donnant pour chaque élément de `L` le nombre de fois qu'il apparait.
2. Écrire une fonction `plus_frequent(L: list, k: int) -> str` qui renvoie le mot de `k` lettres qui apparait le plus grand nombre de fois dans la liste de de chaînes de caractères `L`.

Exercice 2

On souhaite pouvoir déterminer si deux listes contiennent les mêmes éléments avec le même nombre d'occurences (mais pas nécessairement dans le même ordre).

1. On propose l'algorithme naïf suivant :

```
def compare_naif(L, M):
    n = len(L)
    N = M.copy()
    for x in L:
        j = 0
        trouve = False
        while j < len(N) and not trouve:
            if N[j] == x:
                trouve = True
                N.pop(j)
            else:
                j = j + 1
        if not trouve:
            return False
    return len(N) == 0
```

   Quelle est sa complexité ?

2. Pour améliorer l'efficacité, on peut commencer par trier chaque liste, puis en comparant élément par élément.

   (a) Pour trier une liste, on propose la fonction `tri` suivante. Quel est cet algorithme de tri ? Quelle est sa complexité ?

```
def inter(G, D):
    res = []
    i, j = 0, 0
    while i < len(G) and j < len(D):
        if G[i] < D[j]:
            res.append(G[i])
            i = i + 1
        else:
            res.append(D[j])
            j = j + 1
    if i == len(G):
        res.extend(D[j:])
    else:
        res.extend(G[i:])
    return res
```

```
def tri(L):
    n = len(L)
    if n < 2:
        return L
    else:
        G = tri(L[:n//2])
        D = tri(L[n//2:])
        return inter(G, D)
```

   (b) Écrire une fonction qui permet de comparer deux listes et qui utilise la fonction `tri`. Quelle est sa complexité ?

3. Proposer une troisième version qui utilise la fonction `occurences` de l'exercice précédent. Quelle est sa complexité ?

Exercice 3

Un chiffre de César est une forme faible de chiffrement qui implique la « rotation » de chaque lettre d'un nombre fixe de places. La rotation d'une lettre signifie de décaler sa place dans l'alphabet, en repassant par le début si nécessaire, de sorte qu'après la rotation par 3, 'A' devient 'D' et 'Z' décalé de 1 est 'A'.
Pour effectuer la rotation d'un mot, décalez-en chaque lettre par le même nombre. Par exemple, les mots « JAPPA » et « ZAPPA », décalés de quatre lettres, donnent respectivement « DETTE » et « NETTE », et le mot « RAVIER », décalé de 13 crans, donne « ENIVRE » (et réciproquement). De même, le mot « OUI » décalé de 10 crans devient « YES ». Dans le film 2001 : l'Odyssée de l'espace, l'ordinateur du vaisseau s'appelle HAL, qui est IBM décalé de -1.

1. Écrire une fonction appelée `cesar(mot : str, i : int)->str` qui prend comme paramètres une chaîne de caractères et un entier, et qui renvoie une nouvelle chaîne qui contient les lettres de la chaîne d'origine décalées selon le nombre donné.

   La fonction interne `ord` convertit un caractère en un code numérique, et `chr` convertit des codes numériques en caractères.

2. Deux mots forment **une paire par rotation** si vous pouvez effectuer la rotation d'un d'entre eux avec un chiffre de César pour en obtenir l'autre.

   (a) Écrire une fonction qui lit les mots dans *mots.txt* et les stocke comme clés dans un dictionnaire. Peu importent les valeurs associées (`None` ou `True` par exemple).

   (b) Écrire une fonction qui prend en argument le dictionnaire précédemment créé et une chaîne de caractère, et qui affiche toutes les rotations de la chaîne présentes dans le dictionnaire.

   (c) Écrire enfin une fonction qui affiche toutes les paires par rotations présentes dans le fichier *mots.txt*.

Exercice 4

L'algorithme de compression de texte LZ78 (dû à Abraham Lempel et Jacob Ziv en 1978) construit et utilise un dictionnaire pour compresser des données. Il les décompresse à l'aide du dictionnaire inversé.

Méthode

- On se donne un texte, `texte`, à compresser.
- On initialise le code avec `code = []`.
- On initialise un dictionnaire : `dico = {'':0}` ; les clés en seront des chaînes de caractères, les valeurs leurs numéro d'insertion dans le dictionnaire.
- On place une fenêtre de longueur un en position `i = 0` au dessus de `texte`.
- On étend la fenêtre d'observation tant que la chaine `w` qui y figure est dans le dictionnaire (et tant que l'on n'atteint pas la fin du texte).
- On ajoute au code le tuple `(p, s)` avec `p` tel que `dico[w] = p` et `s` est le caractère suivant, c'est-à-dire, celui que l'on doit ajouter à `w` pour que `w+s` ne se trouve pas dans le dictionnaire.
- On insère la nouvelle clé `w+s` dans le dictionnaire.
- On réitère le procédé à partir de la nouvelle position `i` qui suit celle de `s` tant que $\cdots$

À l'issue du procédé, `code` et `dico` permettent de reconstituer le texte. En pratique, pour des données d'une certaine taille la place prise par `code` et `dico` est nettement inférieure à celle de `texte`.

Par exemple, on souhaite compresser le texte `'veridique ! dominique pique nique en tunique.'`.
Initialement `dico = {'':0}` et `code = []`

Lu		Ajout dans le dictionnaire	Ajout dans la liste
v	non trouvé	v 1	(0, v)
e	non trouvé	e 2	(0, e)
r	non trouvé	r 3	(0, r)
i	non trouvé	i 4	(0, i)
d	non trouvé	d 5	(0, d)
i	trouvé en position 4		
iq	non trouvé	iq 6	(4, q)
u	non trouvé	u 7	(0, u)
e	trouvé en position 2		
e␣	non trouvé	e␣8	(2,␣)
!	non trouvé	! 9	(0,!)
␣	non trouvé	␣10	(0,␣)
d	trouvé en position 5		
do	non trouvé	do 11	(5, o)
m	non trouvé	m 12	(0, m)
i	trouvé en position 4		
in	non trouvé	in 13	(4, n)
i	trouvé en position 4		
iq	trouvé en position 6		
iqu	non trouvé	iqu 14	(6, u)
e	trouvé en position 2		
e␣	trouvé en position 8		
e␣p	non trouvé	e␣p 15	(8, p)
i	trouvé en position 4		
iq	trouvé en position 6		
iqu	trouvé en position 14		
ique	non trouvé	ique 16	(14, e)
␣	trouvé en position 10		
␣n	non trouvé	␣n 17	(10, n)
i	trouvé en position 4		
iq	trouvé en position 6		
iqu	trouvé en position 14		
ique	trouvé en position 16		
ique	non trouvé	ique␣18	(16,␣)
e	trouvé en position 2		
en	non trouvé	en 19	(2, n)
␣	trouvé en position 10		
␣t	non trouvé	␣t 20	(10, t)
u	trouvé en position 7		
un	non trouvé	un 21	(7, n)
i	trouvé en position 4		
iq	trouvé en position 6		
iqu	trouvé en position 14		
ique	trouvé en position 16		
ique.	non trouvé	ique. 22	(16, .)