TD IPT^2 N° 6: DICTIONNAIRES 1:
PREMIÈRES UTILISATIONS ÉLÉMENTAIRES, TABLE DE HACHAGE, COLLISIONS

EXERCICE N°1: Algorithme de compression par dictionnaire: l'algorithme LZ78

Les algorithmes de compression par dictionnaire travaillent en général sur un fichier source qui est une suite de symboles de type chaine de caractères. Ils produisent en retour **une liste de tuples** `code`**, accompagnée d'un dictionnaire** `dc`.

Supposons que nous ayons un texte `texte` à compresser. Le principe est le suivant:

- On initialise `code` avec une liste vide: `code=[]`
- Le dictionnaire est initialisé avec une clé (type chaine de caractère) **vide** et une valeur associée 0:

$$dc = \{"" : 0\}$$

- L'algorithme parcourt `texte`.
- Si le préfixe rencontré w n'est pas dans le dictionnaire, on crée une nouvelle clé avec ce préfixe et sa valeur p sera sa position dans le dictionnaire.
- On poursuit, tant que la chaine de caractères w lue est déjà dans le dictionnaire, et tant que l'on n'a pas atteint la fin du texte
- On ajoute à la liste `code` le tuple:

$$(p, s)$$

  avec p tel que $dc[w] = p$ et s le caractère suivant dans la chaîne.
- On insère alors la nouvelle clé `w+s` dans le dictionnaire
- On itère le processus à partir de la position i qui suit celle de s tant que l'on n'a pas atteint la fin du texte.

Une fois ce processus terminé, `code` et `dc` permettent de reconstituer `texte` par l'algorithme de décompression; si le texte est assez long, alors la taille cumulée de `code` et `dc` est nettement inférieure à celle de `texte`.

❶ On propose l'implémentation suivante pour la compression:
D'abord une fonction `compression(texte))` de parcours du texte:

Listing 1: fonction compression(texte)

```
def compression(texte):
    n=len(texte)
    i,code,dc=0,[],{"":0}
    while i<n:
        code,i=compresser(texte,code,dc,i)
    print(code)
return code, dc
```

Puis la fonction `compresser(texte,code,dc,i)`:

Listing 2: Fonction compresser(texte,code,dc,i)

```
def compresser(texte,code,dc,i):
    n=len(texte)
    assert i<len(texte)
    w=""
    while i<n and w+texte[i] in dc: #on avance sur le préfixe tant que dans le dict.
        w+=.....
        i+=1
    if i<n:
        ###le préfixe w est dans le dictionnaire###
        ###le nouveau préfixe wl=w+texte[i] n'y figure pas et y est ajouté###
        p=len(dc) #la position est donnée par la longueur du dictionnaire
        c=texte[i]
        wl=w+c
        dc[wl]=p
        code.append(......)
    return code,i+1
```

Compléter le code précédent.

❷ On propose le texte de "test" suivant:

ABRACADABRA

Faire tourner "à la main" l'algorithme proposé en langage naturel, et déterminer les contenus de `code` et `dc` en fin d'exécution:

❸ Proposer un code simple permettant de créer un dictionnaire `dci` inversé, i.e. dont les clés sont les valeurs de `dc` et les valeurs ses clés. Ecrire le dictionnaire `dci` correspondant au texte **ABRACADABRA**.

❹ Proposer une méthode permettant, à partir de la liste de tuples `code` et du dictionnaire inversé `dci`, de reconstituer le code de test **ABRACADABRA**.

❺ En déduire un script python `decompression(code:`*`list`*`,dc:`*`dict`*`)` $\longrightarrow$ *`str`* recevant en argument la liste de tuples `code` ainsi que le dictionnaire associés à la compression du texte `texte` et qui retourne `texte`.

EXERCICE N°2: Fonction de hachage - résolution d'une collision

On considère la fonction de hachage: $h(k) = k \bmod 13$ et une table de hachage comportant $m = 13$ alvéoles.

❶ Hacher dans cet ordre les clés 26, 37, 24, 30, et enfin 11. Que se produit-il?

❷ On souhaite résoudre les collisions par la technique dite *d'adressage ouvert et sondage linéaire* dont le principe est le suivant:

- On hache la clé en direction de l'alvéole $h(k)$
- si une collision se produit alors on calcule l'adresse de la nouvelle alvéole avec la fonction:

$$h_i(k) = (h(k) + i) \ mod m$$

avec i le premier entier naturel permettant de pointer vers une alvéole vide.

❸ On souhaite enfin hacher les clés 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Que se passe-t-il? Quelle solution proposer?

EXERCICE N°3: Exploration détaillée d'une collision

Considérons des clés k constituées des caractères de la table ASCII étendue, soit 256 caractères (codés sur 1 octet), et associons à chaque clé un entier correspondant à sa décomposition en base 256; par exemple, on fait correspondre à la clé `"pouet"` constituée des caractères `p, o, u, e, t` de valeurs ASCII respectives `112, 111, 117, 101, 116` l'entier `e` suivant:

$$e = 112 \times 256^4 + 111 \times 256^3 + 117 \times 256^2 + 101 \times 256 + 116 = 482906301812$$

❶ Ecrire une fonction Python de hachage `h(k)` qui prend en argument la clé (chaine de caractères) `k` et renvoie l'entier `e` correspondant selon la méthode ci-dessus. On exploitera pour cela l'écriture des polynômes selon le schéma de Horner:

$$\begin{aligned} P(x) &= a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \ldots a_{n-m} + \ldots + a_1 x + a_0 \\ &= ((((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + a_{n-3}) \ldots) x + a_0 \end{aligned} \quad (1)$$

❷ Ecrire une fonction Python `entier_chaine(e)` qui réalise l'opération inverse. Conclure sur une propriété que possède cette fonction de hachage.

❸ On considère la fonction d'adressage f:

$$\begin{array}{rcccl} f & : & \mathbb{U} & \rightarrow & \mathbb{N} \\ & & k & \mapsto & h(k) \ mod \ 255 = n \end{array}$$

a· Implémenter la fonction `f(k)` en Python d'argument la clé `k`.

b· QUESTION PRATIQUE: appliquer la fonction `f(k)` sur la chaine (clé) `"pouet"`. Faire de même avec les clés `"chariot"` et `"haricot"`; que constate-t-on sur ces deux derniers cas?

❹ En remarquant que:

$256 \equiv 1$ [255]

et que par conséquent $n \times 256 \equiv n$ [255] et finalement $256^k \equiv 1$ [255], expliquer le surprenant résultat de conversion des clés `"chariot"` et `"haricot"`.

❺ On constate donc que la présence de clés "anagrammes" perturbe le fonctionnement de l'indexation. Proposer une fonction python `dico_valide(dict)` qui prend en argument le dictionnaire et vérifie la présence éventuelle de clés anagrammes.

NB: dans tout l'exercice, on pourra exploiter les commandes `ord(char)` et `chr(n)` qui renvoient respectivement le code ASCII du caractère `char` et le caractère correspondant à l'entier `n`.