Correction du TD itc^2 n° 6: Dictionnaires 1:

utilisations élémentaires, table de hachage, collisions

Exercice n°1: Algorithme de compression par dictionnaire: l'algorithme LZ78

❶ On propose le code complété ainsi:

Listing 1: 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+=texte[i]
        i+=1
    if i<n:
        ###le préfixe w est dans le dictionnaire###
        ###le nouveau préfixe w1=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]
        w1=w+c
        dc[w1]=p
        code.append((dc[w],c))
    return code,i+1
```

❷ On obtient à partir du texte "ABRACADABRA":

$$code = [(0,'A'),(0,'B'),(0,'R'),(1,'C'),(1,'D'),(1,'B'),(3,'A')]$$

$$dc = \{'': 0,'A' : 1,'B' : 2,'R' : 3,'AC' : 4,'AD' : 5,'AB' : 6,'RA' : 7\}$$

❸ Le code pour inverser clés et valeurs est élémentaire en procédant par compréhension:

```
dci={nvcle:nvval for (nvval,nvcle) in dc.items()}
```

Le dictionnaire inversé donne pour le texte "ABRACADABRA":

$$dci = \{0 :'', 1 :' A', 2 :' B', 3 :' R', 4 :' AC', 5 :' AD', 6 :' AB', 7 :' RA'\}$$

❹ Principe de reconstruction du texte:

On rappelle les contenus de `code`, `dc` et `dci` :

$$code = [(0,'A'),(0,'B'),(0,'R'),(1,'C'),(1,'D'),(1,'B'),(3,'A')]$$

$$dc = \{'': 0,'A' : 1,'B' : 2,'R' : 3,'AC' : 4,'AD' : 5,'AB' : 6,'RA' : 7\}$$

$$dci = \{0 :'', 1 :' A', 2 :' B', 3 :' R', 4 :' AC', 5 :' AD', 6 :' AB', 7 :' RA'\}$$

- On inverse le dictionnaire `dc` i.e. les valeurs (str) et les clés (int).
- On initialise un texte vide `texte=""`.
- On itère sur `code` (variable `el`);
  - le premier élément du tuple `el[0]` i.e. la position du caractère est extrait et permet de récupérer dans le dictionnaire inversé le préfixe avant le caractère qui va être traité: `dci[el[0]]`.
  - On récupère ensuite le caractère à ajouter après le préfixe avec `el[0]`.
  - On concatène enfin `texte`, le préfixe avant le caractère à ajouter `dci[el[0]]` et le caractère traité `el[1]`

❺ Le code correspondant est le suivant:

Listing 2:

```
def decompression(code,dc):
    dci={nvcle:nvval for (nvval,nvcle) in dc.items()} #
    construction du dictionnaire inversé
    texte=""
    print("Dictionnaire inversé:",dci)
    for el in code:
        pref=dci[el[0]]
        texte+=pref+el[1]
    return texte
```

Le déroulement pas à pas du code donne:

el	w	s	texte
(0,"A")	""	"A"	"A"
(0,"B")	""	"B"	"AB"
(0,"R")	""	"R"	"ABR"
(1,"C")	"A"	"C"	"ABRAC"
(1,"D")	"A"	"D"	"ABRACAD"
(1,"B")	"A"	"B"	"ABRACADAB"
(3,"A")	"R"	"A"	"ABRACADABRA"

Exercice n°2: **Fonction de hachage - résolution d'une collision**

❶ On obtient les hachages suivants:

k		$h(k)$
26	$\rightarrow$	0
37	$\rightarrow$	11
24	$\rightarrow$	11
30	$\rightarrow$	4
11	$\rightarrow$	11

❷ La table de hachage comporte 13 alvéoles numérotées de 0 à 12. Après résolution des collisions par adressage ouvert et sondage linéaire, on obtient l'occupation suivante de la table:

$h_i(k) = (h(k)+i)\,\%m$	k
0	26
1	11 (3 collisions)
2	
3	
4	30
5	
6	
7	
8	
9	
10	
11	37
12	24 (1 collision)

❸ On constate que la taille de la table va être insuffisante pour ajouter 1,2,3,4,5,6,7,8,9,10 :

$h_i(k) = (h(k)+i)\,\%m$	k
0	26
1	11 (3 collisions)
2	**1** (1 collision)
3	**2** (1 collision)
4	30
5	**3** (2 collisions)
6	**4** (2 collisions)
7	**5** (2 collisions)
8	**6** (2 collisions)
9	**7** (2 collisions)
10	**8** (2 collisions)
11	37
12	24 (1 collision)

Les valeurs 9 et 10 ne trouvent plus d'alvéoles libres avec l'adressage ouvert et sondage linéaire.

Deux solutions sont envisageables:

- On augmente la taille du dictionnaire (c'est à dire on augmente m).
- On procède à un chainage (vu en cours).

Exercice n°3: **Exploration détaillée d'une collision**

Listing 3:

1.
```python
def h(k):
    res=ord(k[0])*256+ord(k[1]) #on initialise la valeur de res
     avant la boucle
    for i in range(2,len(k)):
        res=res*256+ord(k[i])
    return res
```

Autre proposition, toujours itérative:

Listing 4:

```python
def h_2(ch):
    for i in range(0,len(ch)-1):
        if i==0:
            res=ord(ch[i])*256+ord(ch[i+1])
        else:
            res=res*256+ord(ch[i+1])
    return res
```

Enfin, le schéma de Horner est particulièrement adapté à une approche récursive:

- la construction de la clé hachée se fait cette fois en partant de la fin (coefficient a_0 calculé en premier); le code renverra le cas récursif avec:

```
return ord(ch[-1])+256*h_rec(ch[:-1])
```

- le cas de base va intervenir lorsqu'il restera uniquement dans la chaine les deux premiers caractères, correspondant aux deux coefficients d'ordre plus élevé a_n et a_{n-1}:

```
return ord(ch[0])*256+ord(ch[1])
```

Cela donne:

Listing 5:

```python
def h_rec(ch): #on part cette fois du dernier caractère
    if len(ch)==2: #cas de base
        return ord(ch[0])*256+ord(ch[1])
    else: #cas récursif
        return ord(ch[-1])+256*h_rec(ch[:-1])
```

Le code précédent ne fonctionne qu'avec des clés de longueur minimale 2; pour corriger cela et envisager des clés à un caractère on peut faire:

Listing 6:

```python
def h_rec(ch): #on part cette fois du dernier caractère
    if len(ch)==1: #cas de base
        return ord(ch)
    else: #cas récursif
        return ord(ch[-1])+256*h_rec(ch[:-1])
```

Listing 7:

2.
```python
def entier_chaine(e):
    res=""
    while e%256!=0:
        res=chr(e%256)+res
        e=e//256
    return res
```

Du fait de la définition de la fonction h, chaque chaîne de caractère possède une image unique dans $\mathbb{N}$, la fonction h est donc injective.

Listing 8:

3.
```python
def f(k):
    return h(ch)%255
```

4. On obtient la sortie suivante:

```
>>> print h("pouet")
47
>>> print h("chariot")
236
>>> print h("haricot")
236
```

5. La valeur obtenue par application du modulo sur l'entier "long" e peut s'écrire:

$$(a_n \cdot 256^n + a_{n-1} \cdot 256^{n-1} + \ldots + a_1 \cdot 256 + a_0 \cdot 256^0)[255]$$
$$= (a_n + a_{n-1} + \ldots + a_1 + a_0)\,[255]$$

L'indice renvoyé est donc indépendant de l'ordre des coefficients $(a_n, a_{n-1}, a_{n-2}, \ldots, a_0)$, et sera donc **le même pour des mots anagrammes**.

6. Script de vérification de validité du dictionnaire pour `f(ch)`:

   NB: cette procédure ne fonctionne que dans le cas d'un dictionnaire dont les clés sont des chaines.

Listing 9:

```
def dico_valide(dict):
    res=True
    listecles=[cle for cle in dict.keys()]
    for i in range(len(listecles)):
        for j in range(i+1,len(listecles)):
            if f(listecles[i])==f(listecles[j]):
                res=False
    return res
```

Enfin, on propose cette variante qui évite un balayage total du dictionnaire par l'emploi de boucles conditionnelles:

Listing 10:

```
def dico_valide_bis(dict):
    res=True
    listecles=[cle for cle in dict.keys()]
    i=0
    while res and i<len(listecles)-1:
        j=i+1
        while res and j<=len(listecles)-1:
            res=f(listecles[i])!=f(listecles[j])
            j=j+1
        i+=1
    return res
```

Cette dernière variante exploite une liste de 256 entiers (d'indice $\in [0, 255]$), tous initialisés à 0. Cette liste va recueillir, à chaque valeur d'indice, le nombre d'occurrences du hachage de clé qui a justement donné la valeur de cet indice (liste de fréquences); pour assurer l'absence d'anagrammes, aucun total d'occurrences ne doit dépasser 1: on teste donc cela directement dans la boucle conditionnelle de balayage de la liste des clés; la complexité est ainsi nettement réduite par rapport aux scripts proposés plus haut:

Listing 11:

```
def dico_valide_ter(dict):
    res=True
    listecles=[cle for cle in dict.keys()]
    tabfreq=[0]*256
    i=0
    while res and i<len(listecles):
        if tabfreq[f(listecles[i])]==0:
            tabfreq[f(listecles[i])]+=1
            i+=1
        else:
            res=False
    return res
```