TP 2 partie 2

Le but de ce TP est de revoir les graphes, se familiariser avec leur représentation sous forme de dictionnaire (liste d'adjacence) ou leur matrice d'adjacence, et aussi de mettre en application les techniques de hachages sur les dictionnaire.

1 Dictionnaire

Un dictionnaire c'est un peu comme une liste mais indicé par des clés :

```
mon_dictionnaire={"cle":    valeur    ,....}
```

On peut utiliser ce qu'on veut comme clé, une chaine, un tuple, ... sauf une liste
Pour effacer une clé, on utilise `del` ou `pop` :

```
>>> del mon_dictionnaire["cle"]
>>> B.pop(cle) # extrait la valeur de la cle et retire le couple
```

On peut aussi créer un dictionnaire à partir d'une liste :

```
>>>L=[(2,3),('a',2),('a','b')]
>>>L
       [(2, 3),  ('a', 'b')]
>>>B=dict(L)
>>>B
       {2: 3, 'a': 'b'}
```

On peut extraire d'un dictionnaire, la clé `keys()`, la valeur `values()` ou en faire une liste.

```
>>>list(B.keys())# liste des cl s
>>>list(B.values())#liste des valeurs
>>>list(B.items())# liste des couples cl  valeurs
```

On peut vérifier qu'une clé est bien dans le dictionnaire avec la commande `in` :

```
>>> cle in B
```

Exercice 1.1 *D'après Mines Ponts 2024 Dans un souci de compression de l'information, il est intéressant de représenter les caractères les plus fréquents par des expressions courtes et de ne plus nécessairement coder avec des codes de longueur constante chaque caractère. Si `s='abaabaca'`, il est possible de coder le caractère 'a' avec 1 bit, et les caractères `'b'` et `'c'` avec 2 bits afin de coder la chaîne s sur seulement 11 bits en tout.*

1. *Proposer une telle représentation en expliquant pourquoi celle-ci pourra être décodée sans ambiguïté. Vous ferez en sorte que la représentation binaire de `'a'` soit inférieure à celle de `'b'`, elle-même inférieure à celle de `'c'`. La représentation précédente emploie la même longueur pour coder les caractères `'b'` et `'c'` alors que le caractère `'b'` est deux fois plus présent que le caractère `'c'` dans la chaîne `s`.*
2. *Il est possible d'aller un cran plus loin et le codage arithmétique présenté dans cette étude permet un gain de compression comme s'il parvenait à représenter un caractère avec un nombre non entier de bits au prorata de sa fréquence d'apparition. Ce principe de compression est notamment utilisé par la norme JPEG2000 de compression des images. Nous ne le présenterons cependant ici que dans le cadre de l'étude de chaînes de caractères.*
   L'objet de cette partie est d'analyser le contenu d'une chaîne de caractères s afin de déterminer : — les caractères utilisés par la chaîne `s` ; — le nombre d'occurrences de chacun.

(a) *Écrire une fonction nommée* `nbCaracteres(c:str,s:str)->int` *qui prend comme argument un caractère* `c`*, une chaîne* `s` *et qui renvoie le nombre d'occurrences (c'est-à-dire le nombre d'apparitions) de* `c` *dans* `s`*. La fonction doit avoir une complexité linéaire en* n*, la longueur de la chaîne* `s`*.*

(b) *Programmer une fonction* `listeCaracteres(s)` *qui renvoie la liste des caractères utilisés à l'intérieur d'une chaîne* `s`*. Tester cette fonction lorsque* `s='abaabaca'`*.*

(c) *En fonction de la longueur* n *de la chaîne et du nombre* k *de caractères distincts dans celle-ci, déterminer la complexité asymptotique dans le pire des cas de la fonction de la précédente. Par exemple pour* `s='abaabaca'`*, on a* $n = 8$ *et* $k = 3$*. On négligera la complexité des* `append` *mais pas celle des tests d'appartenance de la forme* `i in L`*. Autrement dit,* `c in L` *est de complexité linéaire en* `len(L)`*.*

(d) *Programmer une fonction* `analyseTexte(s:str)->dict` *renvoyant le dictionnaire* `R` *dont la clé est constituée des caractères de la liste et les valeurs de leur occurence dans la liste* `s`*. Tester* `analyseTexte('babaaaabca')`*.*

(e) *En fonction de la longueur* n *de* `s` *et du nombre* k *de caractères distincts présents dans* `s`*, (autrement dit* k *est la longueur de* `listeCaracteres(s)`*), donner une estimation de la complexité asymptotique dans le pire des cas de la fonction* `analyseTexte`*.*

3. *Programmer une fonction* `occurence(s)` *prenant comme paramètre le texte* `s` *et renvoyant le dictionnaire de clé les occurrences* `o` *des caractères du* `s` *et comme valeur les listes de caractères d'occurrence* `o`*.*

4. *Proposer un programmer qui renvoie la liste (classée dans l'ordre alphabétique) des listes de caractères de même occurence classé dans l'ordre décroissant de leurs occurrences.*

2 Hachage

Petit rappel sur les dictionnaires :

```
 mon_dictionnaire={cle1: valeur1,........}
 mon_dictionnaire[cle1]
      [out]  valeur1
del mon_dictionnaire[cle1] # efface la couple (cle1,valeur1)
mon_dictionnaire.pop(cle1)
      [out:]  valeur1# renvoie valeur1 et retire le couple de la dictionnaire
B=dict([[(cle1,valeur1),...])# transforme une liste de couple en dictionnaire
ist(B.items()) # transforme dictionnaire en liste de couple
list(B.keys()) # renvoie liste des cles
list(B.values()) #renvoie liste des valeurs
cle1  2 in B
      [out:]   True # Pour verifier que la cle est dans le dictionnaire
```

Exercice 2.1 *On souhaite implémenter une structure élémentaire de table de hachage sans gestion des collisions. On se restreint ici à des clés entières et on utilisera la fonction de hachage définie sur* $\mathbb{N}$ *par*

$$h(x) = \lfloor (ax \mod w)\frac{n}{w} \rfloor$$

w *un entier premier avec* a *qui doit être grand, par exemple* $a = 3^7$*,* $w = 2^6$ *et* $n = 2^3$*.*

La fonction de hachage renvoie alors un entier $m = h(cle)$ *pour une clé* cle*, on stockera la valeur dans la liste* `L_valeurs` *en position* $h(cle)$*.*

Remarquons que dans le cas étudié, un dictionnaire n'est représenté que par un tableau de longueur m *stockant différentes valeurs associées à des clés et None si pas de valeur.*

1. *Quelle sera une bonne valeur de* m *pour notre table de hachage ?*

2. *Écrire en Python la fonction* h *(on prendra soin de vérifier que l'entrée de* h *est un entier positif au moyen d'une assertion). Pour un dictionnaire* D *on construit donc* `L_valeurs` *ainsi :*

```
L_valeurs=[None]*n
for cle in D:
      L[h(cle]=D[cle]
L_cles=list(D.keys())
```

Les clés sont stockées dans une liste `L_cles` *et les valeurs dans* `L_valeurs` *de longueur* m*. Le dictionnaire est alors le couple* `(L_cles,L_valeurs)`*,*

3. *Écrire une fonction* `get_valeur(L_valeur,cle,h)` *renvoyant la valeur associée à* `cle` *si elle est dans la lite* `L_valeurs` *et* `None` *sinon.*
4. *Écrire une fonction* `insere_valeur(L_valeurs,(cle,valeur),h)` *qui insère valeur au bon emplacement de* `L_valeurs` *pour la clé* `cle`*.*

Exercice 2.2 *Une stratégie de résolution des collisions repose sur le chaînage séparé. Cette méthode consiste à placer dans le tableau des valeurs des listes pouvant contenir plus d'un élément. Dans ce cas, on n'y stocke plus uniquement les valeurs mais des listes contenant les couples* `(cle,valeur)` *afin de distinguer des couples de clés qui entrent en collision pour la fonction de hachage. On note* h *la fonction de hachage utilisée et* `D` *ce tableau. Le code ci-dessous fourni un exemple d'implémentation où les clés sont des chaînes de caractères et les valeurs sont des entiers :*

```
dictionnaire = [[],[('bonjour',42)],[],[('a',12),('bazar',21)],[]]
```

`D` *est donc une liste de liste de uplet, éventuelement vide.*

1. *Combien de valeurs différentes peut produire la fonction de hachage utilisée ?*
2. *Que vaut* `h('a')` *?*
3. *Écrire une fonction* `get_valeur(D,cle,h)` *renvoyant la valeur associée à* `cle` *si elle existe et* `None` *sinon.*
4. *Écrire une fonction* `insere_valeur(D,(cle,valeur),h)` *qui insère le couple* `(cle,valeur)` *au bon emplacement de* `D` *pour la clé* `cle`*. Si la clé existe déjà, la valeur correspondante sera remplacée. Si celle ci n'existe pas encore et s'il y a collision, le couple* `(cle,valeur)` *sera placé à la fin de la liste des couples entrant en collision avec celui-ci.*
5. *Écrire une fonction* `est_vide(D)` *qui renvoie* `True` *si le dictionnaire représenté par* `D` *est vide et* `False` *sinon.*

Exercice 2.3 *On se donne comme fonction de hachage, définie pour une chaine de caractère* `x` *comme suit : On associe à cette chaine (composée de* 256 *types de caractères) l'entier* $f(x)$ *en base* 256 *correspondant, par exemple pour* `Blop'`*.*
Comme `'B'` *vaut* 66*,* `'l'` *vaut* 108 *,* `o` *et* `'p'` *respectivement* 111 *et* 112*, le nombre associé est*

$$f(\texttt{'Blop'}) = 66 \times 256^3 + 108 \times 256^2 + 111 \times 256 + 112$$

Pour finir le hachage, $h(x)$ *sera l'entier* $f(x)$ *modulo* 255*, soit dans l'exemple :* 142

1. *Programmer la fonction hachage* `h` *en s'assurant qu'elle agit sur une chaine de caractères, en faisant appel à la fonction* `ord()` *qui renvoie la valeur d'un caractère entre* 0 *et* 256*.*
2. *Programmer une fonction de hachage avec gestion des risque de collision, prenant comme paramètre un dictionnaire.*