CORRECTION TP 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 Crypter un texte

Dans cette première partie nous allons travailler sur un algorithme de cryptage, assez proche du hachage.

Exercice 1.1 *On cherche à crypter un texte* `t` *de longueur* n*, qui sera une liste de* n *entiers compris entre* 0 *et* 25 *(correspondant au 26 lettres de l'alphabet).*

1. *Proposer une fonction* `codage(t,d)` *renvoyant une liste de même taille que* `t` *décaler de* `d` *modulo* 26*. Puis une fonction décodage renvoyant la liste d'origine.*
2. *On part du principe que dans un texte crypter (en français) la lettre e apparait plus fréquement. Proposer une fonction* `frequence(t')` *renvoyant une liste de taille* 26 *dont la case* i *indique le nombre d'apparition de l'indice* i*.*
3. *On suppose le texte crypter avec un décalage* d *inconnu, proposer une fonction* `decodageauto(t)`*, appelant les précédente et proposant un décodage de* t*.*
4. *On améliore le cryptage en proposant un décalage different pour chaque lettre. On se donne une clé, par exemple,* `concours` *c-a-d* $[3, 15, 14, 3, 15, 20, 18, 19]$*, qui transforme* a *en* c*, puis* a *en* o *et ainsi de suite. Donc le premier nombre est décaler de* 3 *modulo 26 puis* 15 *modulo 26 et ainsi de suite jusqu'au huitième, puis on recommence*
   (a) *Proposer la fonction* `codageV(t,c)` *prenant la liste* t*, la clé (sous forme de liste) et renvoyant le texte codé.*
   (b) *Programmer la fonction* `pgcd(a,b)` *renvoyant le pgcd des entiers* a *et* b*.*
   (c) *On suppose un texte* t *(assez long) codé par* t' *et on veut retrouver la clé. problème une même lettres dans* t' *n'est pas forcément la même pour* t*. On va donc rechercher la répétition, non pas d'une lettre, mais de* 3 *lettres et si cette répétition est suffisamment grande (on a des multiple de* k *la longueure de la clé), le pgcd donne la longueure de la clé.*

      Programmer une fonction `pgcdDrep(t',i)`*, prenant comme paramètre le texte crypter* t' *de taille* n*,* i *un indice,* $(0 \leq i \leq n-2$ *) et qui renvoie le pgcd de toutes les distances entre les répétition (dans* t'*) de la séquence* `t'[i],t'[i+1],t'[i+2]` *dans le texte* `t[i+2 :]`*,* 0 *si pas de répétition..*
   (d) *Programmer* `longueur Cle(t')` *qui renvoie la longueur de la clé (on pourra déterminer la complexité en fonction de* n *de ce programme).*
   (e) *Proposer une fonction de décodage.*

1. La fonction `codage(t,d)` renvoyant une liste de même taille que `t` décaler de `d` modulo 26, puis une fonction décodage renvoyant la liste d'origine :

```
def codage(t,d):
      tprime=[]
      for u in t:
            tprime.append((u+d)%26)
      return tprime
def decodage(t,d):
      tprime=[]
```

```
    for u in t:
        tprime.append((u-d)%26)
    return tprime
    # ou aussi:
    # return codage(t,-d)
```

2. Une fonction `frequence(t')` renvoyant une liste de taille 26 dont la case i indique le nombre d'apparition de l'indice i.

```
def frequence(tprime):
    F=[0]*26
    for u in tprime:
        F[u]+=1
    return F
```

3. Une fonction `decodageauto(t)`, appelant les précédentes et proposant un décodage de t.

```
def decodageauto(t):
    F=frequence(t)
    i=0
    m=0
    for i in range(26):
        if F[j]>m:
            i,m=i,F[j]
     d=i-4
     return decodage(t,d)
```

4. (a) Une fonction `codageV(t,c)` prenant la liste t, la clé (sous forme de liste) et renvoyant le texte codé.

```
def codageV(t,c):
    tprime=[]
    p,n=len(c),len(t)
    for i in range(n):
        tprime.append(( t[i]+c[i%p])%26)
    return tprime
```

   (b) Une fonction `pgcd(a,b)` renvoyant le pgcd des entiers a et b.

```
def pgcd(a,b):
    M,m=max(a,b),min(a,b)
    r=M%m
    while r>1:
        r=M%m
        if r==0:
            return m
        M,m=m,r
    return m
```

   (c) Une fonction `pgcdDrep(t',i)`, prenant comme paramètre le texte crypter t' de taille n, i un indice, $(0 \leq i \leq n-2$) et qui renvoie le pgcd de toutes les distances entre les répétition (dans t') de la séquence `t'[i],t'[i+1],t'[i+2]` dans le texte `t[i+2 :]`, 0 si pas de répétition..

```
def pgcdDrep(tprime,i):
    a,b,c=tprime[i],tprime[i+1],tprime[i+2]
    n=lem(tprime)
    d=[]
    for j in range(n-2):
```

```
            if (a,b,c)==(tprime[j],tprime[j+1],tprime[j+2]):
                d.append(j)
        if len(d)==1:
                return 0
        u=d[0]
        for k in range(len(d)-1):
                u=pgcd(u,d[k])
        return u
```

(d) `longueur Cle(t')` qui renvoie la longueur de la clé (on pourra déterminer la complexité en fonction de n de ce programme).

```
def Cle(tprime):
      n=len(tprime)
      multiple=[]
      for i in range(n-2):
              u=pgcdDrep(tprime,i)
              if u!=0 and u!=1:
                  multiple.append(u)
        k=multiple[0]
        for i in range(1,len(k)):
              k=pgcd(k,multiple(i))
        return k
```

(e) Une fonction de décodage, consiste à rechercher d'abord la longueur de la clé et en partant du principe que la lettre e, de valeur 5 est la plus fréquente rechercher la clé :

```
def decodage(tprime):
      n=len(tprime)
      k=Cle(tprime)
      cle=[None]*k

      t=[None]*n
      for i in range(k):
              texte=[]
              for j in range(i,n,k):
                      texte.append(tprime(j))
              texte=decodageauto(texte)
              for j in range(len(texte)):
                      t[j+i*k]=texte[j]
      return t
```

Exercice 1.2 5/2

```
import numpy as np
import numpy.random as rd
def poids(bc):
    return bc//10,bc%10
def entier(b,c):
    return 3*b+c
def poidsfort(x):
    return x//3
def poidsfaible(x):
    return x%3
def convert(t):
```

```
    T=[]
    for x in t:
        T.append(poidsfort(x)*10+poidsfaible(x))
    return T
# t=rd.randint(0,8,10)

# convert(t)
def longueurMotif(t,i,j,m):
    l=0
    m=np.min([len(t)-i,m,len(t)-j])
    while l<=m:
        if t[i+l]!=t[j+l]:
            return l
        l+=1
    return l
def longueurMotifmax(t,i,j,m):
    assert j>i
    M=0
    for k in range(j-i):
        l=longueurMotif(t, i+k, j, m)
        if l>M:
            M=l
    return M
def motifmax(t,i,j,m):
    #global A,L,C
    assert j>i
    M,K=0
    for k in range(j-i):
        l=longueurMotif(t, i+k, j, m)
        if l>M:
            K,M=k,l
    A,L,C=M,K,t[j+1]
    return A,L,C
```

2 Les arbres

Un arbre pondéré est un graphe auquel on affecte une valeur (poids) à une arrête, le poids d'un chemin est alors la somme des pondérations des arrêtes. On peut alors représenter l'arbre soit comme une matrice A en affectant la pondération de l'arrête reliant le noeud i au noeud j au coefficient $A_{i,j}$ et $+\infty$ si les noeuds ne sont pas reliés. Ou choisir de le représenter avec une liste d'adjacance en choisissant le couple $\{u_i : [(v_1, p_1), \ldots, (v_k, p_k))\}$ ou v_j est un noeud et p_j le poids de l'arrête reliant le noeud u_i au noeud v_j.

Exercice 2.1 *Proposer un programme prenant comme paramètre la matrice d'adjacence d'un graphe et renvoyant un dictionnaire (ou une liste) de liste d'adjacence.*

```
def listeadj(A):
      n,p=np.shape(A)
      D={}
      for i in range(n):
            u=[] # les liste de noeuds adjacent a i
            for j in len(p):
                  if A[i,j]=!0:
                        u.append(j)
            if len(u)>0:
```

```
                D[i]=u
        return D
```

Des commandes de tris :

```
L.sort() # tri par ordre croissant
L.reverse() #  change l'ordre
min(L) # ressort le min
max(a) # je vous laisse deviner
L.insert(i,a)# insert le terme a en position i
del L[i] # efface le terme en position i
L[i:j] # extrait la sous liste d'indice i a j-1
L.remove(a)# retirer l'element a de L
L.count(i)# indique le nombre de terme de L de valeur i
```

Définition 2.1 *Colorer un graphe connexe non orienté, c'est associer une couleur à chaque noeud de telle sorte que deux noeuds adjacents soient de couleur différentes. Le nombre chromatique d'un graphe est le nombre minimal nécessaire de couleurs pour colorer un graphe. Il est inférieur à $\Delta+1$ où Δ est le degré maximum des noeuds du graphe.*

On propose deux algorithmes permettant d'obtenir une coloration d'un graphe et donc une "idée" très approchée de ce nombre chromatique :

Exercice 2.2 *Algorithme Welsh Powell*

Etape 1 *Ranger les sommets dans l'ordre décroissant de leur degré*

Etape 2 *Choisir une nouvelle couleur dite courante et colorer ainsi le premier sommet de la liste non coloré*

Etape 3 *Dans la liste des sommets, colorer avec la couleur courante, les sommets de la liste non encore colorés qui ne sont pas adjacents avec un noeud coloré avec cette couleur.*

Etape 4 *Si tous les sommets ne sont pas encore colorés revenir à l'étape 2*

1. *Programmer la fonction* **`deg(A,i)`** *prenant pour paramètre la matrice d'adjacence A qui n'a que des 1 et des 0, l'entier n et retournant un entier correspondant au degré du noeud n.*
2. *Programmer la fonction* **`listeNoeud_degre(A)`** *retournant une liste de liste* **`[n,deg(n)]`** *du couple n, le numéro du noeud, et $deg(n)$ son degré.*
3. *Programmer la fonction* **`ordonnernoeud(A)`** *prenant pour paramètre la matrice d'adjacence du graphe et retournant la liste des noeuds ordonner dans l'ordre décroissant de leur degré.*
4. *On défini* **`couleur`** *la liste composée de n entiers où* **`couleur[i]`** *est la couleur du noeud i, est un entier non nul s'il est coloré et nul s'il ne l'est pas. Programmer une fonction* **`nbnoeud(L)`** *prenant pour paramètre une liste d'entier et retournant le nombre d'éléments nuls (noeuds non colorés).*
5. *Proposer une fonction prenant pour paramètre la matrice d'adjacence d'un graphe et retournant l'entier n indiquant le nombre de couleurs utilisées par l'algorithme de Welsh Powell en complétant ce qui suit :*

```
def welshpowell(A):
    L=ordonnernoeud(A)
    n=len(A)
    couleur=[0]*n # la couleur 0 indique que le noeud n'est pas colore
    c=0 # pas de couleur
    while nbnoeud(couleur)>0:
        c=c+1# La nouvelle couleur Etape 2
        .........
    return c
```

```
def deg(A,n):
    d=0
    for i in range(len(A)):
        d=A[n][i]+d
    return d
# ou aussi:
def deg(A,n):
      d=0
      for u in A[n]:
           d+=u
 # Et m me
 def deg(A,i):
 return sum(A[i])
def listeNoeud_degre(A):
       L=[]
       n=len(A)
       for i in range(n):
               L.append([i,deg(A,i)])
       return L
 # mais aussi:
 def listeNoeud_degre(A):
       D=listadj(A)
       for u in D:
            L.append([len(D(u)),u])
       return L
def ordonnernoeud(A):
    n=len(A)
    listedeg=listeNoeud_degre(A)
    while len(listedeg)>0:
        degnoeud,numeronoeud=listedeg[0]
        j=0
        for i in range(len(listedeg)):
            if listedeg[i][0]>numeronoeud:
                degnoeud,numeronoeud=listedeg[i]
                j=i
        noeudord.append(numeonoeud)
        del listedeg[j]
    return noeudord
# sinon pour les pro de python:
def ordonnernoeud(A):
      listedeg=listeNoeud_degre(A)

       listedeg.sort(reverse=True)
       return listedeg
def nbnoeud(L):
    S=0
    for a in L:
        if a==0:
            S=S+1
    return S
def welshpowell(A):
    adj=listadj(A)
    L=ordonnernoeud(A)
    n=len(A)
```

```
    couleur={}
    c=0
    while len(couleur)<n:
        u=L[0]
        del L[0]
        attent=adj[u[1]]
        c=c+1
        couleur[u[1]]=c
        for v in L:
            if v[1] not in attent:
                L.remove(v)
                couleur(v[1])=c
                attent=attent+adj[v[1]]
    return c, couleur
```

Ce premier algorithme n'est pas franchement folichon, on en propose un autre.

On considère un graphe $G = (V, E)$ simple connexe et non-orienté. Pour chaque sommet v de V, on calcule le degré de saturation $DSAT(v)$ de la manière suivante :

$DSAT(v)$ = **nombre de couleurs différentes dans les sommets adjacents à v.**

Exercice 2.3 *L'algorithme* `DSATUR` *est un algorithme de coloration séquentiel, au sens où il colore un seul sommet à la fois et tel que :*

Etape 1 *Au départ le graphe n'est pas coloré.*

Etape 2 *Ordonner les sommets par ordre décroissant de degré.*

Etape 3 *On colore un des sommets de degré maximum avec la couleur* 1*.*

Etape 4 — *Choisir un sommet non coloré avec DSAT maximum. En cas d'égalité, choisir un sommet de degré maximal.*
— *Colorer ce sommet par la plus petite couleur possible.*

Etape 4 *On stoppe l'algorithme* `DSATUR` *quand tous les sommets de G sont colorés sinon retour à l'étape 4.*

1. *Programmer la fonction* `DSAT(A,couleur,i)` *prenant pour paramètre la matrice d'ajacence A la liste des couleurs des noeuds du graphe et l'entier i et retournant le DSAT(i).*
   On pourra compléter le programme suivant :

```
def DSATUR(A,couleur,i):
    adj=[]
    for j in range(len(A[i])):
        if A[i][j]!=0:
            adj.append(j)
    dsatur=0
    ..............
    ..............
    ..............
    return dsatur
```

   Que représente la liste `adj` *?*

2. *Proposer un programme, découpé en plusieurs fonctions permettant de retourner le nombre de couleur nécéssaire pour la coloration DSATUR.*

```
def DSATUR(A,couleur,i):
    adj=[]
    for j in range(len(A[i])):
        if A[i][j]!=0:
            adj.append(j)
```

```
    dsatur=0
    while len(adj)>0: # triter tous les noeuds adjacents
        u=adj.pop()
        c=couleur[u]
        dsatur=dsatur+1 # une nouvelle couleur

        for j in adj:
            if couleur[j]==c:
               adj.remove(j) # on retir tous les noeuds de la meme couleur

    return dsatur
 # Ou aussi:
 def DSATUR(A,couleur,i):
    adj=[]
    for j in range(len(A[i])):
        if A[i][j]!=0:
            adj.append(j)
    c=[0] # liste des differentes couleurs
    for k in adj:
         u=couleur[k]
         if  u not in c: # si u est une nouvelle couleur
             c.append(u)
    dsatur=len(c) -1
    return dsatur

def deg(A,n):
        return sum(A[n])
def ordonnernoeud(A):
   L=[]
   for k in range(len(A)):
        L.append([deg(A,k),k])
   L.sort(reverse=True)
   return [ u[1] for u in L]
def DSAT_max(L,couleur,A):
      d=0 # dsat max
      for i in noeud:
            dsatur=DSAT(A,couleur,i)
            if couleur[i]==0 and dsatur>d:
                d=dsatur
                ndsatmax=i # noeud de dsat max
       return ndsatmax
 def couleur_min(couleur,A,j):
       C=[] # liste des couleur adjacente

        for i in range(n):
            if A[j][i]!=0:
                C.append(couleur[i])
         c=1
         while c in C: # choisir la coloration
              c+=1
          return c
def colorDSATUR(A):
    noeud=ordonnernoeud(A)
    n=len(A)
    couleur=[0]*n
```

```
    couleur[noeud[0]]=1
    del noeud[0]
    while couleur.count(0)>0:# tant que tous les noeuds ne sont ps colores
        ndsatmax=DSAT_max(noeud,couleur,A )
        noeud.remove(ndsatmax)
        c=couleur_min(couleur,A,ndsatmax)
        couleur[ndsatmax]=c
    return couleur
```

3 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 3.1 *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. Dans l'exemple précédent, 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`.*

   `1 01 1 1 01 1 10 1`

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`.*

```
def nbCaracteres(c,s):
    o=0
    for u in s:
        if u==c:
            o+=1
    return o
```

(b) *Programmer une fonction* `listeCaracteres()` *qui renvoie la liste des caractères utilisés à l'intérieur d'une chaîne*

```
def listeCaracteres(s:str):
    listeCar = []
    n = len(s)
    for i in range(n):
        c = s[i]
        if not(c in listeCar):
            listeCar.append(c)
    return listeCar
```

Tester cette fonction lorsque `s='abaabaca'`.

(c) $O(nk)$

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

```
def analyseTexte(s:str):
    R = {}
    l = listeCaracteres(s)
    for c in l:
        R[c]= nbCaracteres(c, s)
    return R
```

Tester `analyseTexte('babaaaabca')`.

(e) $O(2nk)$

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`.

```
def occurence(s):
    D={}
    R=analyseTexte(s)
    for u in R:
        v=R[u]
        if v in D:
            D[R[u]].append(u)
        else:
            D[R[u]]=[u]
    return D
```

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.*

```
L=[]
D=occurence(s)
for u in D:
    for v in D[u]:
        L.append([u,v])
L.sort()
M=[]
for u in L:
    M.append(u[1])
```

```
def tri(s):
    L=[]
    D=occurence(s)
    for u in D:
        D[u].sort()
        L.append([u,D[u]])
    L.sort()
    M=[]
    for u in L:
        M.append(L[1])
    return M
```

5. **Exercice 3.2** *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 $n = 2^3$.

```
a,w,n=3**7,2**6,2**3
def h(x):
    assert type(x)==int
    return int(((a*x)% w)*n/w)
def get_valeur(L_valeurs,cle,h):
    u=h(cle)
    return L_valeurs[u]
def insere_valeur(L_valeurs,(cle,valeur),h):

    return L_valeur[h(cle)]=valeur
```

Exercice 3.3 *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. 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)],[]]
```

1. *Combien de valeurs différentes peut produire la fonction de hachage utilisée ? 5*
2. `h('a')`*=3 ?*
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* `L_valeurs` *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.*

```
def get_valeur(D,cle,h):
      j=h(cle)
      u=D[j]
      for k in D[j]:
           if k[0]==cle:
              return k[1]
       return None
def insere_valeur(D,(cle,valeur),h):
      j=h(cle)
      u=D[j]
      if len(u)==0:
         D[j]=[(cle,valeur)]
         return D
      else:
          for s in range(len(D[j])):
           if D[j][s][0]==cle:
              D[j][s]=(cle,valeur)
              return D
           D[j].append((cle,valeur))
```

Exercice 3.4 *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.*

```
def h(x):
     asssert type(x)==str
     y=0
     n=len(x)
     for in range(n):
          y+=ord(x[i])*256**(n-i)
      return y//256
 def hachage(D):
       assert type(D)==dict
       H={}
       for u in D:
            c=h(u)
            if c in H:
               H[c].append(D[u])
            else:
               H[c]=[D[u]]
        return H
```