CHAPITRE II
Les graphes, dictionnaires et hachage

1 Listes et dictionnaires

1.1 Les listes.

Une liste de taille n est un n-uplet (une suite, un vecteur) numérique ordonné. Concrètement, il s'agit d'une adresse à laquelle est attachée une suite de valeurs numérotées de 0 à $n-1$; sa longueur étant n. Plusieurs opérations et fonctions peuvent être appliquées sur les listes : la longueur `len(...)`, la concaténation `+` et la suppression de valeurs `del(...)`

```
>>> L=[2,3,4,7]
>>> L[1] # la valeur d'indice 1 de la liste... donc la deuxieme valeur
3
>>> len(L)# la longueur de la liste
4
>>> [3,4]+L # on parle de concatenation
[3,4,2,3,4,7]
>>> L=2*L
>>> L
[2,3,4,7,2,3,4,7]
>>> del L[2:5] # efface de la liste les valeurs d'indices 2 inclus a 5 exclu
>>> L
[2, 3, 3, 4, 7]
>>> L=[1,2,3,6,7]
>>> L[1:4] # sous liste de L d'indices de 1 inclus a 4 exclu
[2,3,6]
>>> L=[1,2,3,4,5]
>>> L=L[0:2]+L[3:5] # On retire la valeur d'indice 2
>>> L=[1,2,4,5]
>>> 2 in L
True
>>> 7 in L
False
>>> a,b,c,d=L
>>> a
1
>>> d
5
>>> L[0]=3 # Modifier la premiere valeur de la liste
>>>L
[3,2,4,5]
```

Pour construire une liste de termes non affectés, on utilise `None` :

```
>>> L=5*[None] # On repete par concatenation 5 fois [None]
>>> L
[None, None, None, None, None]
```

```
>>>[] # creer une pile
>>>p.pop() # retire  le dernier terme de la liste et le retourne
>>>p.append(v) # ajoute v comme dernier terme dans la liste
>>>len(p) # retourne la taille de la pile
>>>p[-1]# retourne le sommet de la pile.
```

On peut aussi dépiler et empiler à gauche....
Pour ça on préfèrera importer `deque()` du module `collections` :

```
>>>from collections import deque
>>>A=deque()
>>>A
       deque([])
>>>B=deque([1,2,3])
>>>B
        deque([1, 2, 3])
>>>B.popleft()
      1
>>>B
      deque([2,3])
>>>B.appendleft('a')
>>>B
      deque(['a', 2, 3])
```

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
```

1.2 Les dictionnaires.

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

```

>>> mon_dictionnaire={"mot de passe":   123   ,"identifiant":"Bozo"}
# "123" et "Bozo" sont indexe par "mot de passe" et "identifiant"
>>> mon_dictionnaire[mot de passe]
123
>>> mon_dictionnaire["mot de passe"]=456 # change valeur de  mot de passe
>>> mon_dictionnaire[mot de passe]
456
```

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["mot de passe"]
>>>B= {2: 3, 'a': 'b'}
>>> B.pop(2)
       3
>>> B
       {'a': 'b'}
```

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())
       [2, 'a']
>>>list(B.values())
         [3, 'b']
>>>list(B.items())
          [(2, 3), ('a', 'b')]
```

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

```
>>> 2 in B
        True
```

Exercice 1.1 *D'après CINP 2024 : L'awalé se joue à deux. À tour de rôle, les joueurs prennent les graines situées dans un trou de leur rangée pour ensuite les déplacer dans les autres trous. Des graines peuvent ensuite être récoltées pour que les joueurs se constituent une réserve personnelle. L'objectif est de récolter plus de graines que son adversaire. Au départ, le plateau est com- posé de 2 rangées de 6 trous contenant chacun 4 graines. Le jeu s'arrête si l'un des joueurs obtient dans sa réserve personnelle à côté du plateau plus de la moitié des graines en jeu, c'est-à-dire au moins 25 graines ou jusqu'à une situation empêchant le gain de nouvelles graines. Les 2 participants jouent à tour de rôle. Chaque coup consiste à choisir une case non vide de son camp, à prendre toutes les graines de cette case en main et à les semer à raison d'une graine par case en suivant le sens direct. À la fin de son tour de jeu, la case de départ choisie par le joueur est nécessairement vide. Lorsqu'un joueur n'a plus de graines dans son camp, son adversaire est obligé de jouer un coup qui lui en apporte au moins une.*

Par ailleurs, il est interdit de jouer un coup qui ôte, après récolte, toutes les graines du camp adverse.

La récolte consiste à retirer les graines du plateau pour les stocker dans sa réserve personnelle sur la table à côté du plateau de jeu. Le joueur récolte les graines disponibles après son tour de semence, en commençant par la dernière case dans laquelle il a semé et sous les conditions suivantes :

Une fois qu'un joueur a terminé de semer, il peut récolter les graines du plateau de jeu (en respectant la règle ci dessous).

- *la case appartient au camp adverse (condition 1) ;*
- *cette case contient exactement 2 ou 3 graines (condition 2) ;*
- *s'il vient de ramasser les graines de la case, le joueur doit continuer la récolte dans le sens inverse de la semence, si la case respecte les deux premières conditions ;*
- *il est interdit d'affamer son adversaire, on ne peut donc pas prendre toutes les graines du camp adverse. Si la phase de récolte se termine ainsi, alors la récolte est annulée (condition 3 liée à la règle de la famine).*

1. *Programmer une fonction `initialisation(nom_1,nom_2)` prenant comme paramètre deux chaines de caractères et renvoyant un dictionnaire `jeu` de clé : `'joueur1', 'joueur2','score','n','plateau'` et de valeur respective les nom des joueurs, le tuple des scores initiaux, le nombre de tour effectués initialement, l'état du plateau sous forme de liste de liste.*
2. *Ecrire une fonction `joueur(jeu)` qui renvoie le numéro du joueur dont c'est le tour de jeu.*
3. *Proposer un programme `copie(jeu)` qui renvoie une copie du dictionnaire `jeu`.*
4. *Proposer une fonction `deplacer_graines(plateau:[int],case:tuple)` modifiant en place l'argument plateau. On rappelle que l'on ne sème pas de graine dans la case choisie en début du tour. La fonction `deplacer_graines(plateau: case: tuple)` prend comme argument le `plateau` de jeu ayant la structure choisie précédemment et la case `ligne,colonne)` non vide où le joueur actuel prend les graines. Cette fonction réalise la prise des graines de la case choisie et les sème une par une dans le sens direct (sens inverse des aiguilles d'une montre). Elle renvoie les coordonnées de la case où la dernière graine a été semée.*
5. *Écrire une fonction auxiliaire `case_ramassable(plateau:[int], case:tuple)` qui renverra `True` si le joueur dont c'est le tour a le droit de ramasser les graines de la case proposée et `False` sinon. On ne testera pas la question de la famine pour simplifier le problème. Pour rappel, le joueur peut ramasser le contenu de la case si :*
   - *la case appartient au camp de l'adversaire, soit toujours dans la deuxième moitié du plateau ;*
   - *la case contient 2 ou 3 graines.*
6. *Écrire la fonction `ramasser_graines(plateau:[list], joueur:[int])`, prenant comme paramètre le plateau et le numéro du joueur. Cette fonction utilisera la fonction précédente `case_ramassable` et devra modifier le plateau en place et renvoyer le résultat de la récolte des graines.*
7. *Il faut vérifier à chaque tour de jeu si le choix d'une case est autorisé ou non. Si c'est un joueur humain qui joue, son choix peut se porter sur une case "interdite", c'est-à-dire dont il ne peut pas prendre les graines. Si c'est un joueur virtuel, celui-ci doit pouvoir faire la liste des cases "acceptables". Une case est "acceptable" si :*
   - *condition 1 : elle est du côté du joueur dont c'est le tour ;*
   - *condition 2 : elle est non vide ;*

— *condition 3 : à la fin du tour de jeu, les cases de l'adversaire ne sont pas complètement vides (condition de famine).*

Écrire la fonction `cases_possibles(jeu:dict)` *qui renvoie la liste des indices de toutes les cases jouables par le joueur actif. Le dictionnaire jeu ainsi que sa clé plateau ne devront pas être modifiés.*

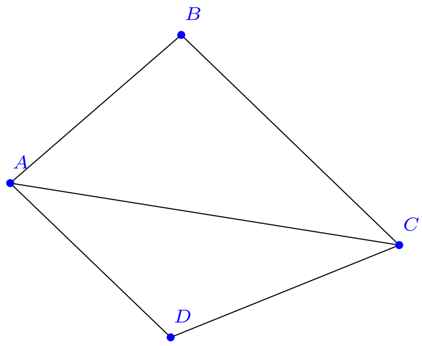
2 Notion de graphe, de chemin, graphe étiquetés et matrice d'adjacence

2.1 Généralités

Définition 2.1 *Un graphe G non orienté est un couple (S, A), où S est un ensemble (de sommets ou noeuds) et A un ensemble de paires d'éléments de S (appellées arêtes). Une chaine, reliant le sommet a à b, est une suite d'arrête du type :*

$$\{a, e_2\}\{e_2, e_3\}....\{e_n, b\}.$$

L'ensemble des noeuds reliés à a forme une composante connexe du graphe et on dit qu'un graphe est connexe si tous ses noeuds sont reliés les uns aux autres.
Deux noeuds sont dits adjacents s'il existe une arrête les reliant.
Le degré d'un noeud a, est le nombre d'arête contenant a.
Le nombre de noeuds d'un graphe n est l'ordre du graphe, à noter qu'un graphe d'ordre n admet au plus $\binom{n}{2}$ arrêtes.



On remarque que :

$$\deg(A) = 3$$

On peut aussi orienté un graphe, on parle alors de graphe orienté. Dans ce cas on distingue degré entrant (nombre d'arrêtes menant au noeud) et degré sortant (nombre d'arrêtes partant du noeud) Si un noeud est relié à lui même on parle de cycle, et un graphe connexe sans cycle est un arbre.

Définition 2.2 *Un arbre est un graphe connexe sans cycle, on peut associer une donnée à chaque noeud (une étiquette), et l'ordonnée en désignant une racine r, avec comme convention $u > v$ si le noeud u appartient au chemin reliant r à v.*

Pour représenter les graphes finis (A est un ensemble fini) et faire des opérations, on utilise.... nos amies les matrices :

Définition 2.3 *La matrices d'adjacence d'un graphe de n noeuds, est la matrice carré A de $M_n(\mathbb{R})$ ou n est le nombre de sommet où a_{ij} indique le nombre d'arêtes reliants le noeud i au noeud j. Si le graphe est non orienté, la matrice est symétrique. Dans le cadre d'arbres pondérés a_{ij} prend la pondération de l'arrête reliant le noeud i au noeud j.*

Pour le graphe précédant :

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

On remarquera que si le graphe n'est pas orienté la matrice est symétrique. Cependant la représentation matriciel n'est pas forcément économique (dans le cas des graphes qui ont peu d'arrête ça fait beaucoup de 0) en mémoire, on peut donc choisir de la représenter avec une liste d'adjacence, c'est à dire une liste des noeuds adjacents à chacun des noeuds.
Pour le graphe précédent :

```
 >>> L=[[2,3,4],[1,3],[1,2,4],[1,3]] # avec une liste
 >>>G={'A':['B','C','D'],'B':['A','C'],'C':['B','A','D'],'D':['A','C']} # avec un dictionnaire
>>>G['B']
       ['A', 'C']
```

Le premier noeud (A le noeud 1) est relier aux noeuds de `L[0]`. Ainsi `L[i]` est la liste des noeud relier au noeud i.

Définition 2.4 *Un chemin de longueur p est une chaine constituée de p noeuds. Un cycle est un chemin reliant un noeud à lui même.*

Exercice 2.1 *Recherche du nombre de chemin de longueur n Soit G un graphe fini (non pondéré), la longueur d'un chemin reliant un noeud est le nombre d'arêtes composant le chemin.*

1. *Soit A la matrice d'adjacence du graphe justifier que $a_{i,j}^k$ le coefficient de la i ème ligne j ième colonne de la matrice A^k indique le nombre de chemin de longueur k reliant le noeud i au noeud j.*
2. *Proposer un programme, indiquant le nombre de chemin de taille k reliant le nœud i au nœud j.*

2.2 Parcours de graphe

-Parcours en profondeur : A partir d'un noeud, passer à un de ses voisins et ainsi de suite, et s'il n'y a pas de voisin revenir au précédent. On affiche les noeuds visités et on obtient la liste des noeuds accessible à partir du noeud d'origine.

-Parcours en largeur A partir d'un noeud on explore tous les voisins puis tous les voisins de voisins jusqu'à ce qu'on l'ai déjà rencontrer. On affiche les noeuds visités

En iteratif (on verra un programme recursif dynamique plus tard)

```python
def Parcours_prof(G,S):
"""  G un graphe dictionnaire, S un sommet, renvoie la liste des sommets visites"""
assert type(G)==dict and type(S)==str
visite=[]# liste des voisins visites
marque={}# dictionnaire voisins visites
attente=deque() # pile sommets en attente
attente.append(S)
while len(attente)>0:
        u=attente.pop()
        if u not in marque:
           visite.append(u)
           marque[u]=True
           for s in G[u]:
                if s not in marque:
                     attente.append(s)
return visite
```

En recursif pour un graphe sans **cocycle** :

```python
def Parcours_prof_rec(G,S):
 """ G un graphe dictionnaire, S un sommet, renvoie la liste des sommets visites"""
 assert type(G)==dict and type(S)==str
 if len(G[S])==0:
    return []
 visite=[]
 for s in G[S]:
       visite+=[s]+Parcours_prof_rec(G,s)
 return visite
```

En largeur

```python
def Parcours_largeur(G,S):
"""  G un graphe dictionnaire, S un sommet, renvoie la liste des sommets visites"""
assert type(G)==dict and type(S)==str
visite=[]# liste des voisins visites
marque={}# dictionnaire voisins visites
attente=deque() # pile sommets en attente
attente.append(S)
while len(attente)>0:
        u=attente.popleft()
        if u not in marque:
           visite.append(u)
           marque[u]=True
```

```
            for s in G[u]:
                if s not in marque:
                    attente.append(s)
return visite
```

Exercice 2.2 *Proposer une fonction prenant comme paramètre la liste d'adjacence d'un graphe de type dictionnaire tel que*

1. *Vérifier un graphe est connexe*
2. *Vérifier que deux noeuds sont relier*
3. *Vérifier l'existence de cycle ou non.*

2.3 Algorithme de Diejstra

Soit G un graphe pondérer, orienté ou non, et A la matrice d'adjacence dont les coefficient $a_{i,j}$ corresponde à la distance (la pondération entre le noeud i et j). On cherche à obtenir le chemin le plus court reliant le noeud 0 au noeud i. On propose l'algorithme suivant dit de Diejstra.

Etape 1 . Placer les sommets du graphe dans la première ligne d'un tableau.

. Sur la deuxième ligne écrire le coefficient 0 sous le point de départ et le coefficient ∞ sous les autres sommets.

Etape 2 . Repérer le sommet X de coefficient minimal.

. Commencer une nouvelle ligne et rayer toutes les cases vide sous X.

Etape 3 . Pour chaque sommet adjacent à X calculer la somme p du poids de X ajouter au poids de X à Y.

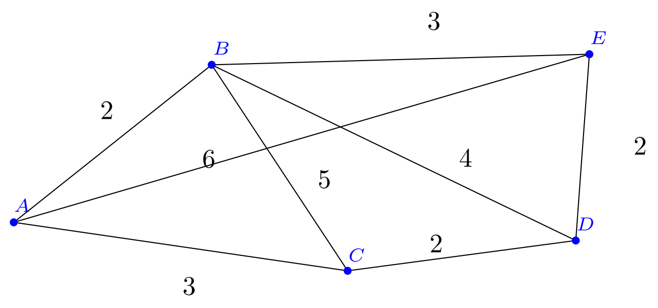
. Si cette somme p est inférieur au poids de Y inscrire le couple (p, X) dans la case de la colonne Y, p est le nouveau poid de Y.

. Si cette somme p est supérieur reporter la valeur de la case du dessus.

. Compléter les autres cases vides (noeuds non adjacents à X) par la valeur de la cases du dessus.

Etape 4 S'il reste des sommets non exploité retourner à l'étape 2.

Etape 5 La longueur minimale est le nombre lu sur la dernière ligne du tableau.



Qui nous donne l'algorithme suivant

A	B	C	D	E
0	2_A	3_A	∞	6_A
$-$	2_A	3_A	6_B	5_B
$-$	$-$	3_A	5_C	5_B
$-$	$-$	$-$	5_C	5_B
$-$	$-$	$-$	$-$	5_B

Ainsi le chemin le plus court pour aller de A vers E est de longueur 5 et est ABE.

1. Soit D un tableau de n lignes 2 colonnes, dont les termes de la première colonne sont des flottants, et ceux de la deuxième colonne des uplets. Programmer une fonction $extraire(D, noeud)$ prenant pour paramètre le tableau D et une liste d'entier et retournant le premier indice i appartenant à la liste noeud tel que $D[i][0]$ soit minimal.
2. La matrice A dont les coefficients $a_{i,j}$ indiquent la distance du noeud i au noeud j ($1e+308$ quand deux noeuds ne sont pas reliés). On construit une fonction diejstra(A) prenant pour paramètre la matrice d'adjacence pondérée A et retournant les chemins les plus courts reliant le noeuds 0 au noeud i sous la forme d'un tableau D de n lignes 2 colonnes, dont les termes de la première colonne sont des flottants, indiquant la distance la plus courte reliant le noeud 0 au noeud i et ceux de la deuxième colonne des listes indiquant le chemin. C'est à dire :

```
>>>D[2]
[24,[0,1,2]]
```

indique que le chemin le plus court du noeud 0 au noeud 2 est $(0, 1, 2)$ et est de longueur 24. Compléter le corps principal du programme.

```
def diejstra(A):
  """A la matrice d'adjacence"""
  n=len(A)
  # creer la liste de noeuds
  noeud=[i for i in range(1,n)]

  #creer le tableau Etape 1
  D=[None]*n
  for i in range(n):
      D[i]=[A[0][i],[0,i]]


  while len(noeud)>0:
      a=extraire(D,noeud)
     ..........
     ............
  return D
```

3 Hachage

Faire des recherches dans un dictionnaire (s'il est de grande taille) peut prendre du temps, et comme on a vu qu'il existe des techniques pour accélérer la recherche si la liste est triée. Il peut être interessant de "indicer" intelligemment les éléments d'un dictionnaire. Comme la "classement" dépend de la clé, on peut imaginer modifier cette clé, par exemple pour la transformer en entiers (si possible pas trop grand) , gagner de la mémoire et organiser le tri plus facilement.

On peut aussi, en tant qu'utilisateur, être amener à vérifier qu'un élément appartient à un dictionnaire (par exemple un individu est il à Decour, est ce un prof un étudiants....) sans être autoriser à connaitre la liste complète (une sorte de de cryptage des données). Pour que ça fonctionne il faut que le hachage (la nouvelle adresse) permette d'identifier exactement le bon individu.

3.1 Fonction de hachage

Un fonction de hachage est une application d'un ensemble E, auquel appartient les clé, vers $\{0, \dots, n-1\}$, ou n est un entier fixé.

Une bonne fonction de hachage serait une application sans collision, c'est à dire qu'à chaque élément de E on associe un unique entier. Comme E est a priori de cardinal bien plus grand que n... ça n'est pas toujours possible.

Le but est donc d'éviter un maximum de collisions.

Voici quelques exemples simples de hachage :

— Pour tout x de $E \subset \mathbb{N}$, $h(x) = \bar{x}$ ou $\bar{x}$ est le reste modulo n :

```
def h(x):
    return x%n
```

— Pour tout x de $E \subset \mathbb{N}$:

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

Où $\overline{ax}$ est le reste modulo w et w un entier premier avec a qui doit être grand, par exemple $w = 2^{22}$ et $n = 2^{11}$

```
def h(a,x):
    return int(a*x%2**22)*2**-11)
```

Le hachage nous permet alors de "ranger" les éléments du dictionnaire dans une liste L à n éléments :

```
def hachage(D,n,h):
""" D un dictionnaire avec des cles entieres"""
     L=[None]*n
     for u in D.keys():
             L[h(u)]=D[u]
      return L
```

3.2 Collision et chainage

Dans le cas d'une collision, c'est à dire deux clés qui ont même hachage, on peut tout simplement "ranger" les valeurs à la même place. Ce qui a du sens si on souhaite par exemple classer des noms par ordre alphabétique en regroupant tous ceux commençant par la même lettre.

```
def hachage(D,n,h):
""" D un dictionnaire avec des cles entieres"""
     L=[[] for u in range(n)]
     for u in D.keys():
             L[h(u)].append(D[u])
      return L
```

Pour renvoyer un dictionnaire avec une clé "hachée"

```
def hachage(D,h):
""" D une dictionnaire """
     d={}
     for u in D:
             i=h(u)
             if i in d:
                    d[i].append(D[u])
             else:
                     d[i]=[D[u]]
      return d
```

Exercice 3.1 *Soit* D *un dictionnaire dont les clés sont des tuples. On propose la fonction de Hachage renvoyant la somme. Programmer cette fonction.*

En python, un dictionnaire est en fait un tableau, les lignes sont triées (numéroté) et finie (un entier n fixé). Pour une clé donnée on applique une fonction de hachage modulo n qui nous donne le numéro de la ligne où on "stockera" la valeur et la clé. La recherche est alors facilitée, car les clés sont finalement triées (recherche dans un tableau trié). En terme de mémoire, c'est évidement plus lourd qu'une simple liste de couples (le tableau a une taille prédéfinie), mais plus maniable. On ne peut donc pas choisir un n trop grand et les problèmes de collisions (deux clés qui ont le même hachage) sont géré par la méthode du chainage.

3.3 Hachage en Python

Python propose une fonction de hachage `__hash__()` pour n'importe quel objet et renvoie un entier de 64 bits en complément à deux (un entier entre -2^{63} et $2^{63}-1$). Qui donne par exemple pour deux chaines de caractère, en apparence très semblable :

```
>>>'bonjour'.__hash__()
     -5528643876924486141

>>>'bonjour!'.__hash__()
     -1073295473186426518 # tres different
```

Qui permet de créer un petit programme hachage rapide :

```
def hachage(D,n):
""" D un dictionnaire """
     L=[[] for u in range(n)]
     for u in D.keys():
             i=u.__hash__()
             L[i].append(D[u])
      return L
```

Ou encore a partir d'une liste :

```
def hachage(L,n):
""" L une liste """
     D=[[None] for u in range(n)]
     for u in L:
             i=u.__hash__()
```

```
            D[i].append(L[u])
      return D
```

Comme n peut etre grand on peut aussi renvoyer un dictionnaire :

```
def hachage(D):
""" D une dictionnaire """
     d={}
     for u in D:
            i=u.__hash__()
            if i in d:
                  d[i].append(D[u])
            else:
                   d[i]=[D[u]]
      return d
```

Exercice 3.2 *1. Soit la fonction f définie sur $\mathbb{N}$ par $f : x \mapsto (x+2) \bmod 10$ où $a \bmod b$ désigne le reste de la division euclidienne de a par b. Montrer que f est une fonction de hachage valide en précisant sur quel ensemble ainsi que le nombre n de hachés différents produits.*

2. *Déterminer l'ensemble des entiers naturels qui sont en collision avec $x = 1$. On construit une fonction de hachage plus complexe g définie sur $\mathbb{N} \cup \mathcal{A}$ où $\mathcal{A}$ est l'ensemble des chaînes de caractères (représentées par des objets de type `str`, du texte...). On note `pos(mot)` la position de la première lettre de mot dans l'alphabet*

   `pos('abeille')=0, pos('bazar')=1 ...`

   g est définie par la relation :

   $$\forall x \in \mathbb{N} \cup \mathcal{A}, \quad \begin{cases} g(x) = f(x) & x \in \mathbb{N} \\ g(x) = pos(x) + 10 & sinon \end{cases}$$

   La command `ord()` renvoie une valeur numérique pour un caractère :

   ```
   ord("a")-97
   Out[1]: 0

   ord("a")-96
   Out[2]: 1
   ```

   Finalement `ord()-97` renvoie la position alphabétique d'une lettre. Programmer la fonction `pos()` prenant comme paramètre une chaine de caractère et renvoyant la position alphabétique de la première lettre. Puis programmer la fonction `g`.

Exercice 3.3 *Dans ce problème, on souhaite stocker des paires (clés,valeurs) en stockant les clés dans une liste (`L_cles`) et les valeurs dans une autre (`L_valeurs`). Un dictionnaire devient donc une paire (`L_cles,L_valeurs`) telle que les éléments `L_cles[i]` et `L_valeurs[i]` forment une paire (clé,valeur). On dispose ici d'une fonction f qui permet de représenter chaque clé par un nombre entier différent (f n'est pas une fonction de hachage au sens où celle-ci fournit toujours des images différentes pour deux clés différentes, il n'y a donc pas de risque de collision). La liste `L_cles` est supposée initialement triée selon les `f(cle)` croissants.*

1. *La fonction `sort()` de Python permet d'ordonner une lisste efficacement :*

```
L.sort(key=g, reverse=False)
# la cle key est une fonction de tri
# reverse=True, decroissant
#par exemple:
def g(u):
     return u[1]
L=[[1,2],[4,1]]
L.sort()
L
Out[4]: [[1, 2], [4, 1]]
 L.sort(key=g)
L
Out[9]: [[4, 1], [1, 2]]
 L.sort(key=g, reverse=True)
L
Out[11]: [[1, 2], [4, 1]]
```

Soit f une fonction d'ordre sur les cle, programmer une fonction prenant comme paramètre un dictionnaire et renvoyant la liste `[Cle,valeurs]` ordonnée par f.

2. *Compléter la fonction `get_valeur(cle,dictionnaire,f)` qui renvoie la valeur associée à `cle` si cette clé est dans dictionnaire et None sinon. Les clés sont triées par `f(cle)` croissants. Quelle est la complexité de cette fonction (en fonction du nombre n de clés du dictionnaires), on supposera que la fonction h s'exécute en $O(1)$.*

```
def get_valeur(cle,dictionnaire,f):
    cles,valeurs = dictionnaire
    gauche ,droite = 0, len(cles) - 1
    while (droite - gauche) > .........:
        milieu = (gauche+droite)//2
        if cles[milieu] == cle:
            return .............
        if f(cles[milieu]) > .................. :
            ........ = milieu
        else:
            ........ = milieu
    if cles[milieu] == cle:
        return ................
    else:
        return ......................
```

3. *On souhaite désormais insérer un nouveau couple `((cle,valeur)` dans le dictionnaire. Pour cela on peut utiliser une fonction similaire à `get_valeur` pour calculer l'indice auquel `cle` et valeur doivent être insérés avec une fonction `indice_insertion(element,dictionnaire,f)` puis écrire une fonction `inserer_dico(element,dictionnaire,f)` qui renvoie le dictionnaire après insertion du couples `(clé,valeur)` contenu dans élément à sa position indicé par f. Implémenter ces fonctions.*

4. *Quelle est la complexité de la fonction précédente dans le meilleur et dans le pire cas ? Préciser si le fait de travailler avec une liste triée selon les clés apporte une amélioration pour le problème d'insertion d'un couple `(clé,valeur)` dans un dictionnaire.*