II Listes, dictionnaires

3 juillet 2024

1. Rappels sur les listes

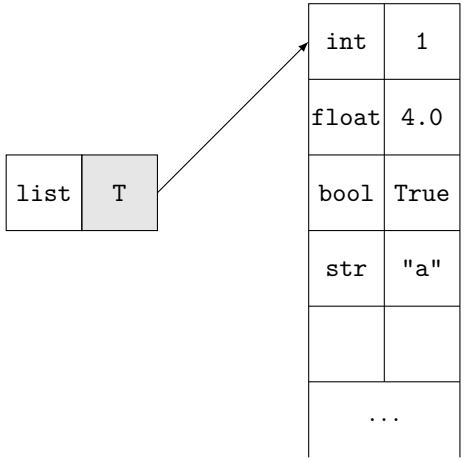
1.1. Représentation mémoire, alias, copie

On peut voir une liste (formellement, un tableau dynamique) comme un tableau d'adresses consécutives. À la création du tableau, tout le tableau n'est pas occupé par les objets de la liste : une place est conservée pour les ajouts ultérieurs. Au besoin (et rarement), le tableau est recopié et agrandi. Des informations sont conservées en plus dans la structure de données (par exemple : le nombre d'éléments).

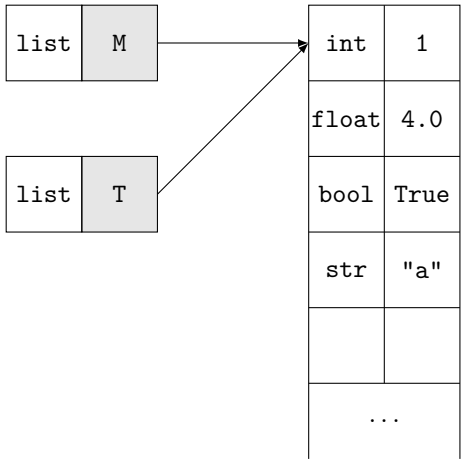
Enfin, on peut modifier une liste par différents mécanismes (réaffectation, ajout par `append` par exemple). Une liste est donc un objet *mutable* : ce qu'elle représente peut varier au fur et à mesure de l'exécution du programme.

Exemple 1.1.1.
Voici la manière dont on pourrait représenter en mémoire le tableau créé par la commande suivante.

```
T = [1, 4.0, True, "a"]
```



Si l'on ajoute la commande `M = T`, on aboutit à la représentation suivante.

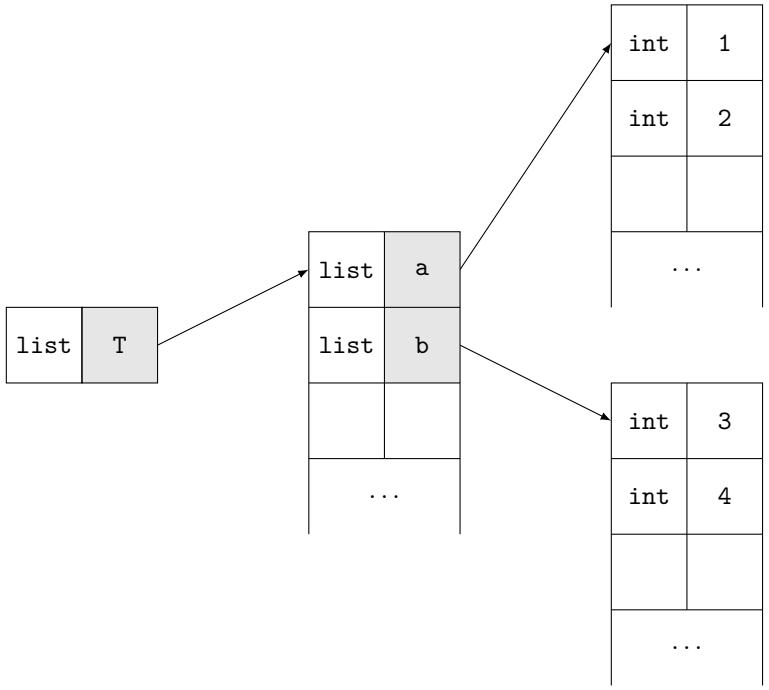


Ici, `M` n'est pas une copie de `T`, mais un *alias* du même objet. Notamment, si l'on modifie `T` (par exemple : `T[0] = 42`), alors `M` est aussi modifié (d'un certain point de vue, `M` et `T` sont les mêmes objets).

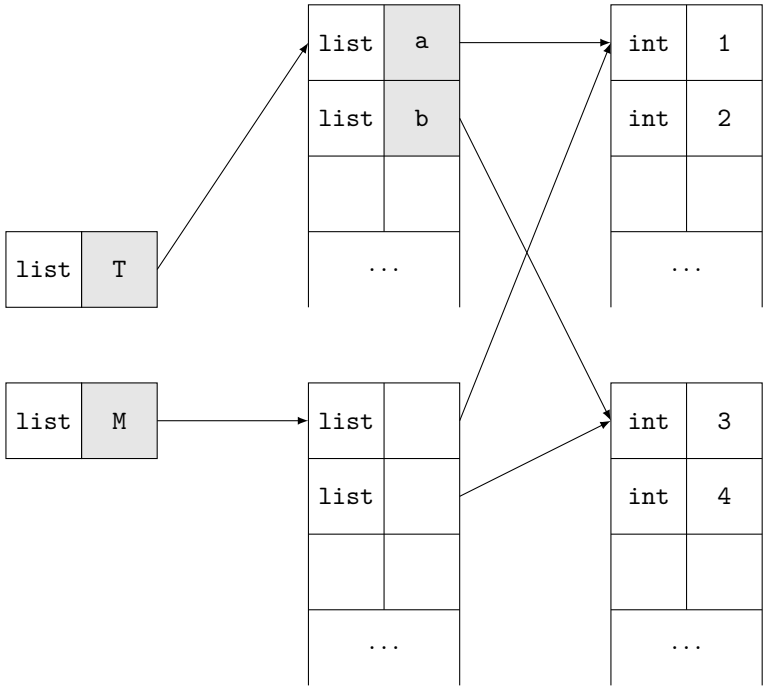
Pour copier `T`, on pourra utiliser : `M = T.copy()`. Toutefois, cela ne résout pas tous les problèmes.

Exemple 1.1.2.
Voici la manière dont on pourrait représenter en mémoire le tableau (matrice) obtenu par les commandes suivantes.

```
1 a = [1,2]
2 b = [3,4]
3 T = [a,b]
```



La commande `M = T.copy()` aboutira à la représentation suivante.



Si l'on modifie `a` (ou `T[0]`, c'est pareil), par exemple par la commande `T[0][0] = 42`, alors `M` sera aussi modifié. On n'a donc réalisé qu'une copie *superficielle*, *i.e.* de la première couche de données de `T`.

Pour réaliser une copie *profonde*, on utilise ceci.

```
4 from copy import deepcopy
5 M = deepcopy(T)
```

1.2. Parcours

Il existe deux manières de parcourir une liste : on peut parcourir directement ses éléments, ou bien parcourir ses indices (ce qui permet bien entendu de parcourir ensuite ses éléments).

Le parcours des éléments d'une liste `L` se fait dans un bloc du type suivant.

```
6 for element in L :
7     # instructions en fonction de element
```

Le parcours des indices d'une liste `L` se fait dans un bloc du type suivant.

```
for i in range(len(L)) :
    # instructions en fonction de i
```

Remarque 1.2.1.
Il est important de choisir le type de parcours adapté en fonction de la question posée. En pratique, on se posera toujours la question « ai-je besoin de connaître les indices des éléments de la liste pour [...] ? ». En cas de réponse positive, on écrira un parcours sur les indices. Sinon, on écrira un parcours sur les éléments.

1.3. Algorithmes types

On liste ici quelques algorithmes simples, que l'on retrouve très (*très*) souvent à l'écrit de l'épreuve d'informatique de la banque PT. Ils sont à connaître sur le bout des doigts. La plupart des algorithmes que l'on vous demandera d'écrire ne seront que des variations des cinq algortithmes ci-dessous.

a. Somme des éléments d'une liste

```
def somme(L) :
    """Précondition : L est une liste de nombres"""
    S = 0
    for x in L :
        S = S + x
    return S
```

b. Calcul du minimum/maximum d'une liste

```
def minimum(L) :
    """Précondition : L est une liste de nombres non vide"""
    mini = L[0]
    for x in L :
        if x < mini :
            mini = x
    return x
```

Exercice 1.3.1.
Écrire une fonction calculant le maximum d'une liste.

c. Calcul du lieu du minimum/maximum d'une liste

```
def argmin(L) :
    """Précondition : L est une liste de nombres non vide"""
    imin = 0
    for i in range(1, len(L)) :
        if L[i] < L[imin] :
            imin = i
    return imin
```

Exercice 1.3.2.
Écrire une fonction déterminant un indice du lieu du maximum d'une liste.

Exercice 1.3.3.
Écrire une fonction prenant en argument une liste de nombres non vide L et renvoyant la liste des indices i vérifiant $L[i] = \min(L)$.

d. Recherche séquentielle d'un élément

```
def recherche(L, e):
    for x in L :
        if x == e :
            return True
    return False
```

e. Comptage du nombre d'occurences d'un élément

```
def compte(L, e):
    nb = 0 # nombre d'occurences de e détectées dans L
    for x in L :
        if x == e :
            nb = nb + 1
    return nb
```

Exercice 1.3.4 (à traiter après avoir revu les dictionnaires)**.**
Écrire une fonction `enumeration` qui prend en argument une liste L d'objets non mutables et qui renvoie le dictionnaire dont les éléments sont les couples de la forme $e : nb$, où

— e désigne un élement de L (sans répétition) ;
— nb désigne le nombre d'occurences de e dans L.

Par exemple, un appel de `enumeration([42, "", 42, True, 42, ""])` renverra le dictionnaire suivant.

```
{42   : 3,
 ""   : 2,
 True  : 1}
```

f. Analyse de complexité

Si l'on note n la longueur de L, alors les cinq algorithmes précédents ont une complexité temporelle dans le pire des cas en $O(n)$.

En effet, on a dans chaque algorithme une boucle, éventuellement précédée d'une instruction élémentaire de complexité $O(1)$. Cette boucle effectue au plus n tours, chaque tour comportant un nombre borné d'opérations élémentaires de complexité $O(1)$. Chaque tour de boucle a donc une complexité en $O(1)$, donc la boucle a une complexité en $nO(1) = O(n)$.

Chaque algorithme a donc une complexité au pire en $O(1) + O(n) = O(n)$.

1.4. Matrices

Nous aurons souvent besoin de représenter des matrices (ou des tableaux multi-dimensionnels) en python. Il est possible de le faire en utilisant des listes de liste. La représentation classique représente une matrice comme une liste de lignes, une ligne étant représentée comme une liste de coefficients. Il est bien entendu possible d'utiliser une autre convention (par exemple représenter une matrice comme une liste de colonnes), même si cela ne sera que rarement le cas.

Exemple 1.4.1.
La matrice $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ sera donc représentée en python par l'objet suivant.

```
41 M = [[1, 2, 3],
42      [4, 5, 6]]
```

Pour une telle matrice `M`, son nombre de lignes s'obtiendra par l'expression `len(M)` et son nombre de colonnes par l'expression `len(M[0])`.

Enfin, on pourra réaliser un parcours directement sur les coefficients par une double boucle imbriquée du type suivant.

```
43 for ligne in M :
44     for coefficient in ligne :
45         # instructions en fonction de coefficient
```

On pourra aussi réaliser ce parcours sur les indices par une double boucle imbriquée, comme suit.

```
46 for i in range(len(M)) :
47     for j in range(len(M[0])) :
48         # instructions en fonction de i (numéro de ligne) et
               ↪ de j (numéro de colonne)
```

Exercice 1.4.2.
On suppose qu'une image en niveau de gris est représentée par une matrice. Un pixel est codé par un entier (0 : blanc, 255 : noir).

Écrire une fonction `image_blanche(n,p)` représentant une image blanche (attention aux alias !) de `n` lignes et `p` colonnes.

Écrire une fonction `nb_pixels_blancs(M)` prenant en argument une matrice M représentant une telle image et renvoyant le nombre de pixels blancs de l'image.

2. Rappels de sup

Les dictionnaires correspondent aux objets de type `dict` en `Python`. Ils forment une structure de données alternative aux listes. Ils permettent d'associer à un premier objet (appelé *clef*) un second objet (appelé *valeur*). Un tel couple clef/valeur est un *élément* du dictionnaire.

2.1. Création d'un dictionnaire

Le dictionnaire vide est l'objet `{}`.

Exemple 2.1.1 (dictionnaire vide)**.**
Taper dans la console les lignes suivantes.

```
49 d={}
50 type(d)
```

La console renvoie alors `<class 'dict'>`.

On peut aussi prédéfinir un certain nombre de couples clef/valeur, avec la syntaxe `clef :valeur,`.

Exemple 2.1.2 (création d'un dictionnaire non vide)**.**
Taper dans la console les lignes suivantes.

```
51 d={3:[],
52    'clef':42,
53    (15,15):[1,2,3,4]}
```

Exemple 2.1.3 (création d'un dictionnaire par compréhension)**.**
Taper dans la console les lignes suivantes.

```
54 d = {str(i) : i for i in range(42)}
```

Les clefs d'un dictionnaire ne sont pas forcément du même type, mais sont toutefois soumises à des limitations de type (nous verrons pourquoi). En pratique, nous utiliserons des clefs entières (type `int`), flottantes (type `float`), des chaînes de caractères (type `str`), ou bien des n-uplets constitués des éléments précédents (type `tuple`).

2.2. Manipulations

L'accès à la valeur d'une clef se fait avec la syntaxe `dictionnaire[clef]`.

Exemple 2.2.1 (accès à une clef)**.**
Taper dans la console les lignes suivantes.

```
55 d={3:[],
56     'clef':42,
57     (15,15):[1,2,3,4]}
58 d['clef']
```

La console renvoie alors `42`.

La demande d'accès à une clef non présente dans le dictionnaire déclenche l'erreur `KeyError`.

Exemple 2.2.2 (erreur `KeyError`)**.**
On reprend l'exemple précédent.

```
59 d[42]
```

La console renvoie alors

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  KeyError : 42
```

On peut tester la présence d'une clef dans un dictionnaire par la syntaxe `clef in dictionnaire`.

Exemple 2.2.3 (test d'appartenance)**.**
On reprend l'exemple précédent. Taper dans la console les lignes suivantes.

```
60 'clef' in d
61 42 in d
```

Ces deux expressions ont respectivement pour valeur `True` et `False`.

On peut affecter à une clef une valeur par la syntaxe `dictionnaire[clef]=valeur`. Si la clef était déjà présente, cela réaffecte `dictionnaire[clef]`. Sinon, cela crée un nouvel élément dans le dictionnaire.

Exemple 2.2.4 (insertion et réaffectation d'un élément)**.**
On reprend l'exemple précédent.

```
62 d[(15,15)] = 1515
```

L'expression `d[(15,15)]` a maintenant pour valeur `1515`.

```
63 d[""] = 0
```

On a créé un nouvel élément dans le dictionnaire, de clef la chaîne vide `""` et de valeur `0`.

Remarque 2.2.5.
Si une valeur est d'un type mutable, on peut alors la modifier sans réaffectation. Tous les exemples précédents montrent bien que les dictionnaires sont des objets mutables en `Python`. Se poseront donc les mêmes problèmes d'alias et de copie que pour les listes.

On pourra utiliser la méthode `.copy()` pour réaliser une copie superficielle d'un dictionnaire, ou bien la fonction `deepcopy` de la bibliothèque standard `copy` pour réaliser une copie profonde de ce dictionnaire.

Exemple 2.2.6 (alias et copies)**.**
Copier et exécuter les scripts suivants (ou bien les rentrer ligne à ligne dans la console.)

```
64 d={0:[]}
65 e=d
66 d[1]=""
67 d[0].append(42)
68 print(e)
```

```
69 d={0:[]}
70 e=d.copy()
71 d[1]=""
72 d[0].append(42)
73 print(e)
```

```
74 from copy import deepcopy
75 d={0:[]}
76 e=deepcopy(d)
77 d[1]=""
78 d[0].append(42)
79 print(e)
```

Enfin, on peut accéder au nombre d'éléments d'un dictionnaire par la syntaxe `len(dictionnaire)`.

2.3. Parcours

On peut parcourir les clefs d'un dictionnaire avec une boucle `for`, avec la syntaxe `for clef in dictionnaire :`.

Exemple 2.3.1.
Imaginons que l'on veuille construire une liste `L` contenant toutes les valeurs d'un dictionnaire `d` (on oublie alors les clefs de ce dictionnaire). On peut procéder comme suit (on crée un dictionnaire arbitraire en début de script, pour les besoins de l'exemple).

```
80 d={0:[],1:42,"":1515}
81 L=[]
82 for clef in d:
83         L.append[d[clef]]
```

On aurait aussi pu écrire directement la ligne suivante, en définissant la liste par compréhension.

```
84 d={0:[],1:42,"":1515}
85 L=[d[clef] for clef in d]
```

Dans les deux cas, la variable `L` a pour valeur `[[], 42, 1515]`.

Remarque 2.3.2.
Étant donné un dictionnaire `d`, on peut accéder à sa liste de clefs par la méthode `keys`, donc avec la commande `d.keys()`, et à sa liste de couples clefs/valeurs par la méthode `items`, donc avec la commande `d.items()`.

On obtient à chaque fois un objet de type `dict_keys` et `dict_items` respectivement, qui se comporteront de manière presque similaire à un objet de type `list` (on peut le parcourir, mais toutefois pas accéder à un élément par son indice).

On peut donc parcourir les clefs de `d` avec la syntaxe `for c in d.keys() :`, ou bien directement les valeurs avec la commande suivante.

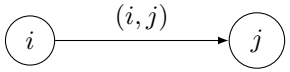
```
86 for v in [it[1] for it in d.items()]:
```

2.4. Exemple important : graphes et dictionnaires d'adjacence

Rappel 2.4.1.
Un graphe orienté est décrit par un ensemble (S, A), où :

— l'ensemble des sommets est S (typiquement, ce sera un ensemble fini) ;

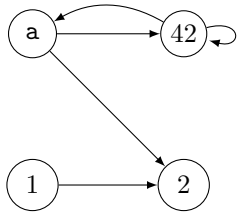
— l'ensemble des arêtes est $A \subset S \times S$: $(i, j) \in A$ s'il y a une arête du sommet $i \in S$ vers le sommet $j \in S$.



Ici, i est un *antécédent* de j et j est un *successeur* de i.

Nous allons souvent décrire des graphes par des dictionnaires d'adjacence. Dans un tel dictionnaire, les clefs sont les sommets du graphe, et la valeur associée à un sommet est la liste des successeurs de ce sommet.

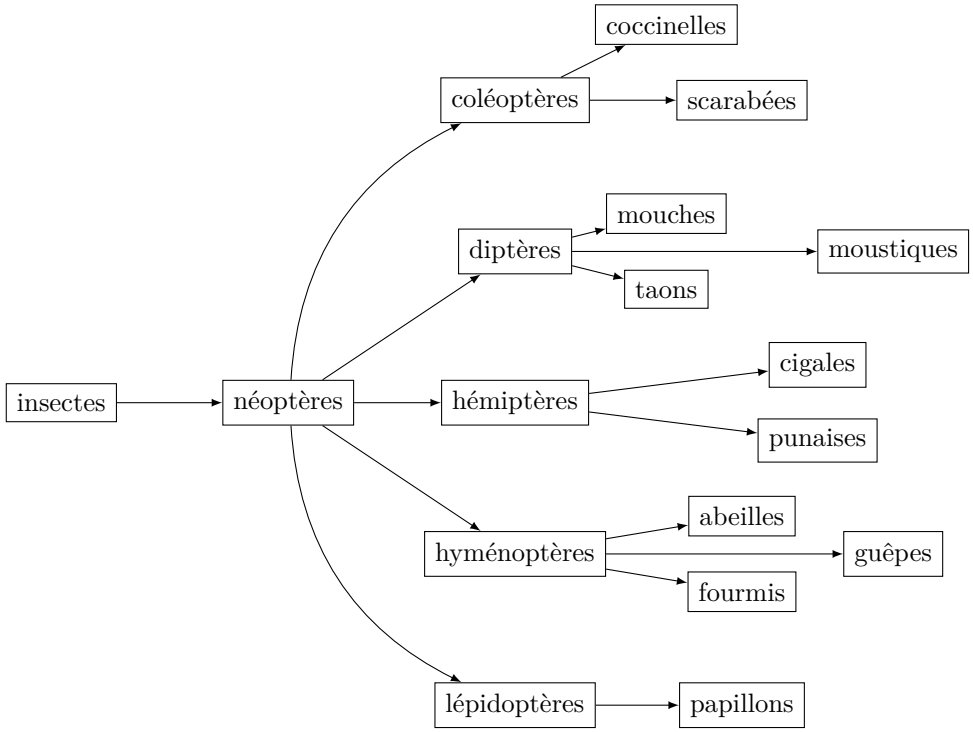
Exemple 2.4.2.
On considère le graphe suivant.



Ce graphe sera représenté par le dictionnaire suivant.

```
87 d = {1  : [2],
88      2  : [],
89      'a' : [2,42],
90      42  : ['a',42]}
```

Exercice 2.4.3.
On donne l'arbre phylogénétique (partiel et simplifié) des insectes.



Créer un dictionnaire représentant ce graphe.

Écrire des instructions ou des fonctions (prenant en argument le dictionnaire d'adjacence d'un graphe) permettant de répondre aux points suivants, et les tester sur notre graphe.

1. Déterminer le nombre de sommets du graphe.
2. Déterminer le nombre d'arêtes du graphe.
3. Déterminer la moyenne du nombre de successeurs par sommet. Que dire de la moyenne du nombre d'antécédents ?
4. Déterminer la liste des sommets du graphe.
5. Déterminer la liste des arêtes du graphe (une arête pourra être représentée par un couple de sommets).
6. Déterminer la liste des boucles (*i.e.* des sommets qui sont leurs propres successeurs).
7. Déterminer la liste des sommets sans successeurs.
8. Déterminer la liste des sommets sans antécédents.

3. Fonctionnement

3.1. Fonction de hachage

Définition 3.1.1.
Une fonction de hachage est une fonction qui prend des objets (informatiques) en entrée et renvoie en sortie un entier.

Une fonction de hachage n'est pas toujours injective. Deux objets ayant même valeur de hachage créent une *collision*.

Remarque 3.1.2.
En fonction des utilisations (par exemple, en cryptographie), on peut demander à ce que cette fonction vérifie des propriétés supplémentaires. Nous n'en parlerons pas ici.

En `Python`, il existe une fonction de hachage déjà codée : `hash`. Elle prend en argument un objet non mutable (parmi une liste de types), et renvoie donc un entier. En pratique, on l'appliquera sur :

— des entiers (type `int`) ;
— des flottants (type `float`)
— des chaînes de caractères (type `str`) ;
— des n-uplets constitués des éléments précédents (type `tuple`).

3.2. Mécanismes et complexité

Voici comment fonctionne sommairement un dictionnaire en `Python`.

— Lorsqu'un dictionnaire (vide) `d` est créé, un tableau (*i.e.* une liste d'adresses consécutives) `T` est créé, d'une taille `n` donnée.
— Lorsque l'on veut insérer un élément `e` associé à une clef `c` (par l'affectation `d[c]=e`), on calcule `h=hash(c)% n`, et l'on affecte la valeur e à la case d'indice `h` du tableau `T`. S'il y a une collision (*i.e.* si la case contient déjà une valeur), on parcourt les cases suivantes et l'on remplit la première cases non vide du tableau `T`.
— Des mécanismes (cachés pour vous) assurent le fonctionnement de tout ceci : retrouver les valeurs déjà définies lorsque l'on utilise `d[c]`, augmenter la taille du tableau `T` lorsqu'il devient trop rempli, gérer les collisions, les ajouts/suppressions de clefs, *etc.*

Notamment, les clefs utilisables doivent pouvoir être passées en argument de la fonction `hash`, d'où les limitations annoncées précédemment.

En pratique, on pourra supposer les complexités suivantes pour les opérations au programme, pour un dictionnaire `d` contenant `n` clefs (voir table 1).

Opération	Commande	Complexité
Création vide	`d={}`	$\mathcal{O}(1)$
Création avec `n` valeurs	`d={`c_1 `:`v_1`,...,`c_n `:`v_n`}`	$\mathcal{O}(n)$
Accès	`d[c]`	$\mathcal{O}(1)$
Insertion ou modification	`d[c]=v`	$\mathcal{O}(1)$
Parcours	`for c in d :`	$\mathcal{O}(n)$
Copie superficielle	`d.copy()`	$\mathcal{O}(n)$
Test de présence (clef)	`c in d`	$\mathcal{O}(1)$
Nombre d'éléments	`len(d)`	$\mathcal{O}(1)$
Liste des clefs	`d.keys()`	$\mathcal{O}(n)$
Liste des couples clefs/valeurs	`d.items()`	$\mathcal{O}(n)$

Table 1 – Complexités pour les opérations sur un dictionnaire

4. Exercices

4.1. Hachage d'une chaîne de caractères

En `Python`, on dispose de la fonction `ord`, qui prend en argument un caractère et renvoie son point de code unicode (c'est un entier).

Proposer une fonction de hachage prenant en objet une chaîne et renvoyant une valeur dans $[\![0, 2^8[\![$ (de sorte que l'on puisse la coder sur un octet).

4.2. Matrices

Comment peut-on faire pour manipuler des matrices avec `Python` en utilisant un dictionnaire ?

4.3. Matrices parcimonieuses

On s'intéresse ici aux matrices *parcimonieuses*, *i.e.* dont la plupart des coefficients sont nuls. Une telle matrice M de dimensions n, p pourra être codée par un dictionnaire ayant pour couples clefs/valeurs :

— `'dim' :` (n, p)

— (i, j) `:`$M_{i,j}$ pour chaque couple (i, j) tel que $M_{i,j} \neq 0$.

Par exemple, la matrice

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix}$$

sera codée par le dictionnaire

`{'dim' :(2,4) , (1,2) :4}`.

Proposer alors une fonction d'addition de deux telles matrices (de même dimension).

Si la première matrice contient c coefficients non nuls, et la seconde c', quelle est la complexité temporelle de cet algorithme ?