Mathématiques & Informatique - F. Gaunard
`http://frederic.gaunard.com`
Lycée Voltaire, Paris 11e.

3 Dictionnaires et structures de données

Le problème du stockage des données est fondamental en informatique: il s'agit de les ranger de façon efficace, de sorte qu'elles soient rapidement accessibles.

L'objet informatique permettant d'enregistrer des données en mémoire s'appelle une *structure de données*, qui est caractérisée par les opérations qu'elle permet et le coût temporel de celles-ci.
On peut par exemple vouloir connaître le nombre d'éléments que contient la structure de données, accéder à un élément particulier, parcourir tous les éléments les uns après les autres...

☞ On choisit ainsi la structure de données en fonction de ses besoins.

Nous allons ici étudier les principales structures de données *itérables*, et voir comment elles sont implantées en machine.

Définition 1.
En Python, un **objet itérable** est un objet qui peut par exemple être utilisé dans une boucle `for`.
Cela signifie que l'on peut parcourir chaque élément de l'itérable *un par un* et effectuer une action sur chacun d'eux.

Parmi les objets itérables (déjà rencontrées), on trouve les listes, les chaînes de caractères, les objets `range( )`, les tableaux `ndarray`... mais aussi les **dictionnaires**, que l'on va introduire ci-après.

1 Dictionnaires

1.1 Notion de dictionnaire

Les dictionnaires sont des structures de données dans lequel la recherche d'information est efficace qui sont couramment utilisés en informatique.

Définition 2. **Dictionnaire**
Un **dictionnaire** (aussi appelé une *table d'association*) est un type de données associant un ensemble de **clés** à un ensemble de **valeurs**.

Plus formellement, si C désigne l'ensemble des clés et V l'ensemble des valeurs, un dictionnaire est un sous-ensemble D de $C \times V$ tel que pour toute clé $c \in C$ il existe au plus un élément $v \in V$ tel que $(c, v) \in D$. Les éléments de D sont appelés des *associations*.

Par exemple, le dictionnaire `Or_Paris24` ci-dessous contient les *associations* pour certains pays ayant participé au Jeux de Paris 2024 avec le nombre de médailles d'or qu'ils ont remportées. Chaque élément du dictionnaire est un couple (*Pays*, *Nombre de médailles d'or*).

```
Or_Paris24={'USA': 40, 'Chine': 20 , 'Japon': 20, 'Australie': 18, 'France': 16}
```

Le dictionnaire permet alors d'accéder à chaque *valeur* (iici le nombre de médailles d'or) correspondant à chaque *clé* (ici le pays).

Un exemple d'application des dictionnaires est le système des noms de domaine ou DNS (pour *Domain Name System*). Les ordinateurs connectés à un réseau comme Internet possèdent une adresse numérique : en IPv4 par exemple, l'adresse est de la forme `x.y.z.t` où `x`,`y`,`z`,`t` sont des nombres entre 0 et 255 codant chacun un octet[1]. Pour faciliter l'accès aux systèmes associés à ces adresses, un mécanisme **associant** un nom à chaque adresse a été mis en place.
Ce mécanisme utilise un dictionnaire dans lequel les *clés* sont les noms de domaine et les *valeurs* les adresses IP. Par exemple, le serveur du moteur de recherche le plus commun dans le monde est associé à l'adresse `8.8.8.8`.

Propriété 1.
Un dictionnaire D supporte les opérations élémentaires suivantes :

- ✗ *lecture* de la valeur v associée à une clé c présente dans D;
- ✗ *ajout* d'une nouvelle association (c, v) dans D;
- ✗ *suppression* d'une association (c, v) de D;
- ✗ *existence* ou non d'une association (c, v) pour une clé $c \in D$ donnée.

1.2 Syntaxe en `Python`

En Python, on crée un dictionnaire avec la syntaxe

```
{c1:v1, ..., cn:vn}
```

où `c1`, ..., `cn` sont des clefs (nécessairement deux-à-deux distinctes) et `v1`, ..., `vn` les valeurs associées.
Par exemple, `{}` est un dictionnaire vide.
Si `D` est un dictionnaire et `c` une clé, les opérations élémentaires sur les dictionnaires s'obtiennent comme suit :

- ✗ `D[c]` renvoie la valeur associée à la clé `c` si celle-ci est présente dans le dictionnaire, et sinon elle déclenche l'exception `KeyError`.
- ✗ `D[c]=v` ajoute une nouvelle association si la clé `c` n'est pas présente dans le dictionnaire, et sinon elle associe la valeur `v` à la clé `c` (en écrasant donc la valeur précédemment associée).
- ✗ `del D[c]` supprime une association si la clé `c` est présente dans le dictionnaire, et sinon elle déclenche l'exception `KeyError`.
- ✗ `c in D` renvoie un booléen indiquant si la clé `c` est présente dans le dictionnaire ou non.
- ✗ La commande `len(D)` renvoie le nombre de clés présentes dans le dictionnaire `D`.

☞ Aux dictionnaires est associé un type dédié, nommé `dict`.

Exemple 1.
Reprenons notre dictionnaire des médailles d'or ci-avant avec quelques opérations élémentaires.

```
>>> Or_Paris24
{'USA': 40, 'Chine': 20 , 'Japon': 20, 'Australie': 18, 'France': 16}
>>> Or_Paris24['France']
16
>>>Or_Paris24['Ukraine']
KeyError: 'Ukraine'
>>>Or_Paris24['Ukraine']=4
>>>Or_Paris24
{'USA': 40, 'Chine': 20, 'Japon': 20, 'Australie': 18, 'France': 16, 'Ukraine': 4}
>>> del Or_Paris24['Australie']
>>> 'Australie' in Or_Paris24
False
>>> len(Or_Paris24)
5
>>> type(Or_Paris24)
<class 'dict'>
```

[1] Il n'y a donc que $2^{32} \simeq 4,3$ milliards d'adresse IPv4, si bien que IPv6 est en cours de déploiement depuis longtemps déjà... En 2023, le taux d'utilisation mondiale de IPv6 serait de 40%.

Si finalement Léon Marchand gagne une (cinquième) médaille d'or qu'on veut rajouter, il suffit d'écraser l'ancienne valeur par la nouvelle en faisant une simple affectation

```
>>>Or_Paris24['France']=21
>>>Or_Paris24['France']
21
```

La méthode `keys` permet d'extraire les clés d'un dictionnaire, sous la forme d'un objet itérable, qui permet donc ensuite de parcourir le dictionnaire.

Plus simplement, on peut parcourir les clés d'un dictionnaire avec la syntaxe `for x in D`.

Exemple 2.

Toujours avec notre dictionnaire `Or_Paris24` :

```
>>> Or_Paris24.keys()
dict_keys(['USA', 'Chine', 'Japon', 'France', 'Ukraine'])
>>> for x in Or_Paris24.keys( ) :
        print(x)

USA
Chine
Japon
France
Ukraine
>>> for x in Or_Paris24:
...     print(x)
...
USA
Chine
Japon
France
Ukraine
>>> for clé in Or_Paris24.keys( ) :
        print(Or_Paris24[clé])

40
20
20
16
4
```

Enfin, on peut extraire les valeurs d'un dictionnaire sous la forme d'un objet itérable avec la méthode `values`, et on peut extraire les couples *(clé,valeur)* d'un dictionnaire sous la forme d'un objet itérable avec la méthode `items` :

Exemple 3.

```
>>> Or_Paris24.values()
dict_values([40, 20, 20, 16, 4])
>>> for clé, valeur in Or_Paris24.items():
...     print(clé, valeur)

USA 40
Chine 20
Japon 20
France 16
Ukraine 4
```

Exercice 1.

On suppose qu'on dispose de trois dictionnaires `Or_Paris24`, `Argent_Paris24` et `Bronze_Paris24` qui associent à chaque pays son nombre de médailles (de chacun des trois types) lors des Jeux de Paris.
Écrire une suite de commandes qui permet de créer un dictionnaire `Paris24` qui associe à chaque pays le total des médailles remportées.

Exercice 2.

On dispose d'un dictionnaire `telephone={'docteur' : '0636636341'}`.
Expliquer ce que fait la commande

```
telephone['docteur']=telephone['docteur'][0:-2]+'0'
```

1.3 Une application : nombre d'occurrence des éléments d'une liste en Python

Étant donnée une liste `L`, on cherche à compter le nombre d'occurrences de chacun des éléments présents dans cette liste. Le plus commode pour cela est d'utiliser un dictionnaire `D` qui, à chaque élément de la liste, associe son nombre d'occurrence. On propose le code suivant :

```
D={}
for x in L:
    if x in D:
        D[x]=D[x]+1
    else:
        D[x]=1
```

Remarque 1.

Au lieu d'un dictionnaire, on pourrait bien entendu utiliser une liste de couples de la forme `(x,n)` où `x` est un élément de `L` et `n` le nombre d'occurrences de celui-ci dans `L`. Néanmoins l'utilisation de cette liste serait plus lente que l'utilisation d'un dictionnaire, ces derniers étant optimisés pour les quatre opérations élémentaires.

1.4 Temps de recherche dans un dictionnaire v.s dans une liste

La question est de mesurer l'intérêt d'une représentation par un dictionnaire plutôt que par une liste de listes.
Nous allons vérifier que le temps de recherche d'une valeur dans un dictionnaire est le même quel que soit le nombre d'éléments de cette structure (ce qui n'est pas le cas avec une liste).

Pour ce faire, on crée une liste de 10^6 couples (i, i), qu'on mélange avec la fonction `shuffle` de la bibliothèque `random`.
La commande `dict` permet de *convertir* une liste dont les composantes sont des listes à deux éléments en dictionnaire.

```
from random import shuffle

liste = [(i,i) for i in range(10**6)]
shuffle(liste)
dico = dict(liste)
```

La fonction `recherche1` prend en paramètres une liste de tuples `L` et un entier `k` et renvoie le deuxième élément de la sous-liste de `L` dont le premier est égal à `k`.
La fonction `recherche2` prend en paramètres un dictionnaire `D` et une variable `c` et renvoie la valeur associée à la clé `c`.
La fonction `recherche3` fait la même chose sans parcourir tous les termes mais en accédant directement à la clé et profite des améliorations offertes par les dictionnaires.

```
def recherche1(L, k):
    for clé, valeur in L:
        if clé == k:
            return valeur

def recherche2(dico, c):
    for clé, valeur in dico.items():
        if clé == c:
            return valeur
```

```
def recherche3(dico, c):
    return dico[c]
```

Les commandes suivantes servent à mesurer le temps de recherche et représenter graphiquement les résultats.

```
import time
import matplotlib.pyplot as plt

temps_liste=[]
temps_dico1=[]
temps_dico2=[]
for k in range(50):
    t0=time.time()
    recherche1(liste,k)
    temps_liste.append(time.time()-t0)
    t0=time.time()
    recherche2(liste,k)
    temps_dico1.append(time.time()-t0)
    t0=time.time()
    recherche3(liste,k)
    temps_dico2.append(time.time()-t0)

abs=[k for k in range(50)]
plt.plot(abs, temps_liste,label="avec une liste")
plt.plot(abs, temps_dico1,label="avec un mauvais usage du dictionnaire")
plt.plot(abs, temps_dico2,label="avec un bon usage du dictionnaire")
plt.legend()
plt.show()
```

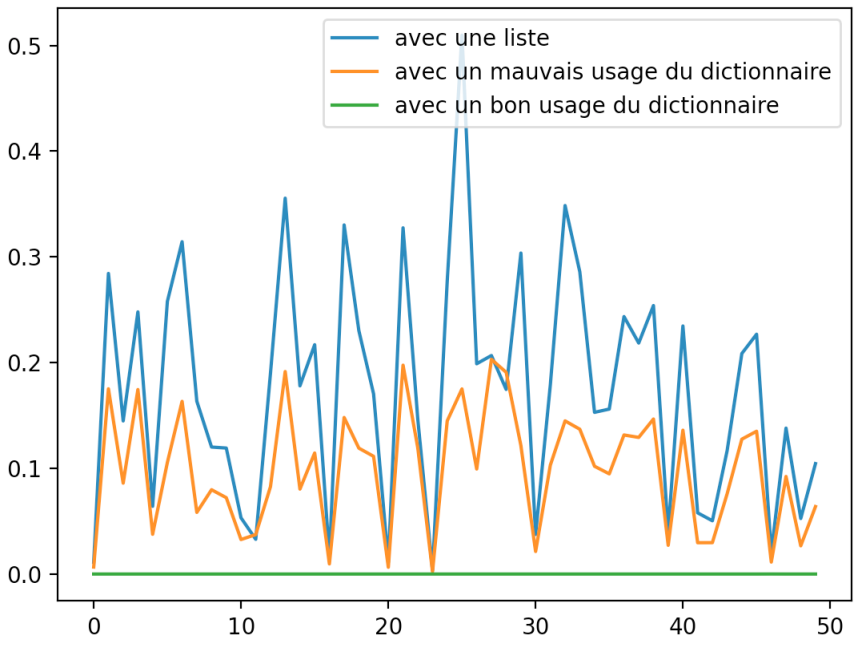

FIGURE 1.

On remarque (FIGURE 1) bien un temps de recherche constant avec une bonne utilisation du dictionnaire.
L'implémentation d'un dictionnaire utilise une *fonction de hachage* f qui permet la recherche $f(\texttt{clé})$ à coût constant.
Ceci n'est pas le cas avec une liste qui présente un temps de calcul qui est fonction de la position de la clé dans la liste.
On enfonce le clou en représentant ((FIGURE 2 et FIGURE 3) l'évolution du temps de recherche d'un élément en fonction

de la taille de la liste. Le code est joint.

```
temps_liste=[ ]
temps_dico1=[ ]
temps_dico2=[ ]

def comparaison(taille):
    liste = [(i,i) for i in range(taille)]
    shuffle(liste)
    dico = dict(liste)
    k = taille/2
    t0=time.time()
    recherche1(liste, k)
    temps_liste.append(time.time()-t0)
    t0=time.time()
    recherche2(dico, k)
    temps_dico1.append(time.time()-t0)
    t0=time.time()
    recherche3(dico, k)
    temps_dico2.append(time.time()-t0)

for puissance in range(1, 8):
    comparaison(10**puissance)

plt.plot(temps_liste,label="avec une liste")
plt.plot(temps_dico1,label="avec un mauvais usage du dictionnaire")
plt.plot(temps_dico2,label="avec un bon usage du dictionnaire")
plt.legend()
plt.yscale('log') # uniquement pour echelle log deuxieme figure
plt.show()
```

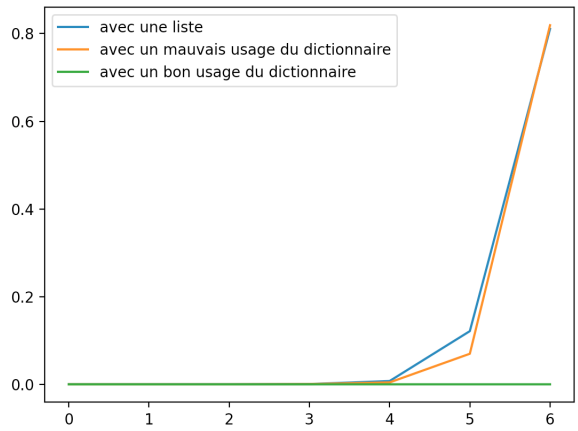

FIGURE 2.

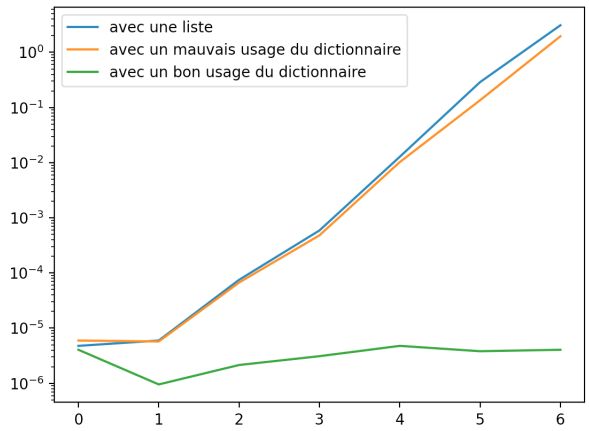

FIGURE 3. Échelle logarithmique

2 Implémentations des dictionnaires et fonctions de hachage

Considérons un dictionnaire D dont on note à nouveau C l'ensemble des clés et V celui de ses valeurs et introduisons $\mathcal{C}$ l'ensemble des clés *potentielles* (que l'on voudrait éventuellement ajouter), de sorte que $C \subset \mathcal{C}$.

Quitte à numéroter les éléments, on suppose dans un premier temps que $\mathcal{C}$ est un intervalle d'entiers (qui peut être très grand[2]).

Une façon naïve d'implémenter un dictionnaire serait d'utiliser un tableau T de taille card($\mathcal{C}$) tel que pour chaque clé $k \in C$, la k-ième case de T, appelée *alvéole*, et notée $T[k]$, "contient" la valeur v associée (ou du moins permet de retrouver cette valeur dans la mémoire). Les cases vides correspondant à des valeurs $u \in \mathcal{C}\backslash C$ sont remplies par une valeur par défaut qu'on note souvent `NIL` ou `NULL`.

Cette implémentation se base donc sur une structure de données plus élémentaire : les tableaux, diminutif de tables à adressage direct. Il n'est pas nécessaire de comprendre en détail comment les tableaux sont eux-mêmes implémentés. Tout ce qu'il faut savoir c'est qu'ils fonctionnent comme des listes de taille fixée en Python (accès à un élément suivant l'indice, modification éventuelle...).

Le problème majeur de cette méthode naïve d'implémentation d'un dictionnaire est qu'en général, comme on l'a signalé ci-avant, l'univers $\mathcal{C}$ des clés est beaucoup trop grand pour pouvoir travailler avec des tableaux de taille card($\mathcal{C}$). Même si les trois opérations *insertion, recherche et suppression* seront très rapides, la mémoire de l'ordinateur ne sera pas suffisante pour manipuler de tels tableaux.
En pratique, C est beaucoup plus petit que $\mathcal{C}$ donc il faut trouver une méthode qui évite le gaspillage de mémoire.

Tables de hachage.

Dans la méthode par adressage direct, une valeur v associée à une clé k d'un dictionnaire est stockée dans l'alvéole k d'un tableau. Le principe d'une **table de hachage** est de stocker v dans la l'alvéole $h(k)$ où $h(k)$ est une *valeur hachée* de la clé. Une fonction de hachage est simplement une fonction

$$h : \mathcal{C} \longrightarrow [\![0, m-1]\!]$$

où l'entier m est en pratique beaucoup plus petit que card($\mathcal{C}$). Une telle fonction ne peut donc pas être injective et ainsi, deux clés distinctes k_1 et k_2 peuvent avoir une même valeur de hachage c'est à dire $h(k_1) = h(k_2)$. On parle alors de **collision**.
Cela entraine que les valeurs v_1, v_2 associées respectivement à k_1, k_2 seront stockées dans la même alvéole.

Pour conduire à une implantation efficace, une fonction de hachage doit :

- ✗ se calculer rapidement;
- ✗ avoir une distribution la plus uniforme possible sur l'ensemble des entiers (soit $[\![-2^{63}, 2^{63}-1]\!]$ sur une machine 64 bits).

La seconde propriété est motivée par le souhait de minimiser le nombre de collisions. Grossièrement, pour tout $i \in [\![0, m-1]\!]$, la probabilité que $h(k)$ soit congru à i modulo m doit être de l'ordre de $\dfrac{1}{m}$.

Résolution des collisions.

Les collisions étant inévitables, une solution dite **résolution des collisions par chaînage** consiste, pour deux clés k_1 et k_2 entrant en collision, à stocker les couples (k_1, v_1) et (k_2, v_2) dans une même alvéole sous la forme d'une liste par exemple.

Pour trouver la valeur associée à une clé k présente dans le dictionnaire, on calcule $h(k)$ pour déterminer l'emplacement du tableau où se trouve la cellule contenant le couple recherché puis on procède à une recherche dans cette alvéole pour trouver le couple qui nous intéresse.
Cette méthode présente l'avantage de pouvoir contenir un nombre p de clefs plus grand que le nombre m de cases du tableau. Si on note $\alpha = \frac{p}{m}$ le taux de remplissage du dictionnaire, les alvéoles auront une taille moyenne de α (sous une hypothèse de hachage uniforme) et la recherche d'une association dans le dictionnaire utilisera un nombre de comparaison en moyenne de l'ordre de α (pour le parcours de chaque alvéole).

Une autre solution consiste, en cas de collision au moment de l'ajout d'une valeur dans le dictionnaire, à chercher un emplacement libre dans lequel déposer la nouvelle association.

[2] rien que si $\mathcal{C}$ est l'ensemble des chaînes d'au plus 4 caractères parmi les 52 minuscules et majuscules, on a déjà un ensemble à 7 454 981 éléments... et si $\mathcal{C}$ est l'ensemble de toutes les chaînes de caractères, on peut considérer que son cardinal est infini !

Cette solution s'appelle une résolution par **adressage ouvert**.

La recherche d'un emplacement libre porte le nom de *sondage*.
Un sondage linéaire consiste à partir de $i = h(k) \mod m$ et à chercher une place libre en testant successivement les cases d'indices $i+1 \mod m, i+2 \mod m, i+3 \mod m, \ldots$ jusqu'à trouver un emplacement libre.
L'inconvénient d'un sondage linéaire est qu'il y a un risque de former des *agrégats*, autrement dit de longues successions de cases contigües occupées, qui nuisent à la répartition uniforme recherchée.

Pour remédier (en partie) à cela, on peut utiliser un sondage quadratique, qui procède de même mais en sondant les cases d'indices $i+1 \mod m, i+1+2 \mod m, i+1+2+3 \mod m, \ldots$
D'autres solutions plus complexes et plus efficaces existent, mais aucune ne résout complètement le problème de la création d'agrégats.

Évidemment, un adressage ouvert exige que le nombre p de clefs soit inférieur à la taille de la table m. Et lorsque le rapport $\alpha = \frac{p}{m}$ se rapproche de 1, il devient de plus en plus difficile de trouver un emplacement vide. Aussi, lorsque α devient trop grand il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances.

3 Application des dictionnaires en programmation dynamique

Les dictionnaires peuvent aussi être utilisés en **Programmation dynamique**. Un autre cours sera complètement dédié à ce principe mais on ne résiste pas au plaisir d'exhiber un exemple d'utilisation dès maintenant.

Un problème posé par la programmation récursive.

Commençons par un exemple classique déjà rencontré dans le TP n°1, celui du calcul des termes de la suite de Fibonacci (F_n) définie par $F_0 = F_1 = 1$ et, pour tout $n \geq 2$, par la relation $F_n = F_{n-1} + F_{n-2}$.
Une première solution : utiliser directement la définition récursive de notre fonction. Cette solution conduit à écrire :

```
def fibo(n):
    if n == 0 or n == 1:
        return 1
    return fibo(n-1) + fibo(n-2)
```

Avec cette fonction le calcul de F_{40} prend environ $0,000021$ secondes et celui de F_{50} plus d'une heure! Ceci s'explique par le fait que le calcul de F_{50} fait appel au calcul de F_{49} et F_{48} mais le calcul de F_{49} fait de son côté appel au calcul F_{48} et F_{47}. Ainsi F_1 est appelé plus de 12 milliards de fois! La complexité est ici exponentielle.

Une seconde solution : utilisation de la mémoïsation.

Le principe de la **mémoïsation** est de conserver les valeurs déjà calculées afin de les réutiliser si nécessaire. Pour cela, on se sert d'un dictionnaire qui va mémoriser le résultat des calculs réalisés de sorte qu'à chaque fois qu'on aura besoin de calculer une valeur, on va voir dans le dictionnaire si elle a déjà été calculée, et dans le cas contraire, on l'ajoutera au dictionnaire. Voici une fonction fondée ce principe :

```
def fibo_bis(n):
    dico = {0 : 1, 1 : 1 } # Un dictionnaire contenant les valeurs de F0 et F1
    if n not in dico: # si le terme Fn n'a pas déjà été calculé
        dico[n] = fibo_bis(n-1) + fibo_bis(n-2)
    return dico[n]
```

Cette approche est descendante avec mémoïsation : elle procède de haut en bas (en anglais : *top-down*), on détermine la valeur d'indice n en redescendant vers les termes d'indices plus petits.

> **Exercice 3.**
> Proposer une fonction `fibo_ter` **non récursive** et ascendante, c'est à dire qui procède de bas en en haut (en anglais : *bottom-up*) toujours avec mémoïsation, qui calcule F_n.

4 Application des dictionnaires aux graphes

En première année a été introduite la notion de graphe. On rappelle qu'un graphe peut être représenté par :

- ✗ une *matrice d'adjacence*, qui est la matrice M dont le terme général $m_{i,j}$ est égal à 1 s'il existe une arête allant de i à j, et 0 sinon. Si le graphe est pondéré, alors $m_{i,j}$ sera l'étiquette de l'arête allant de i à j si elle existe, et 0 sinon.
- ✗ ou une *liste d'adjacence* qui est la liste L, dont le i-ème élément $L[i]$ est la liste des sommets voisins de i.

L'inconvénient de cette représentation est qu'elle impose aux sommets d'être étiquetés par des nombres entiers. Comme ce n'est pas nécessairement le cas, il est plus commode de représenter un graphe à l'aide d'un dictionnaire. Par ailleurs, les dictionnaires étant optimisés pour la recherche d'un élément, il est plus rapide de travailler avec des graphes modélisés par des dictionnaires.

Exemple 4.

Un graphe orienté G_1 (à gauche) et le même graphe pondéré G_2 à droite.


```
D1={'A':['B'],'B':['A','D'],'C':['A', 'B', 'E'],'D':['C'],'E':[ ]}

D2={'A':[('B', 2)],'B':[('A', 3),('D', 5)],'C':[('A', 10), ('B', 6), ('E', 7)],
                              'D':[('C', 4)],'E':[ ]}
```

Ainsi un graphe est représenté par un dictionnaire où les clés sont les étiquettes des sommets, et les valeurs correspondantes sont les listes de voisins (appelées listes d'adjacence). Si le graphe est pondéré, les valeurs correspondantes sont des listes de couples contenant chacun un voisin et l'étiquette de l'arête associée.

4.1 Parcours en largeur d'un graphe

L'algorithme de **parcours en largeur** d'un graphe consiste, partant d'un sommet initial (pris en argument), à visiter tous ses voisins, puis les voisins de ses voisins, etc...

Le principe de l'algorithme est le suivant:

i. mettre le sommet initial dans `FileAVoir`;
ii. retirer le sommet s au début de `FileAVoir` et le marquer comme déjà rencontré;
iii. mettre tous les voisins de s que l'on rencontre pour la première fois à la fin de `FileAVoir`;
iv. tant que `FileAVoir` n'est pas vide, reprendre l'étape *ii.*

La complexité de cet algorithme est en $O(n+p)$, où n est le nombre de sommets et p est le nombre d'arêtes du graphe. En effet, chaque sommet est étudié au plus une seule fois, ce qui se fait en $O(n)$. De plus, à chaque étape, il faut parcourir la liste des voisins du nouveau sommet rencontré : finalement on aura donc parcouru la liste des voisins de chaque sommet, or la somme des longueurs des listes de voisins est en $O(p)$, d'où le résultat.

On peut alors implémenter l'algorithme comme suit, en utilisant un dictionnaire `DejaRencontre` pour enregistrer les sommets déjà rencontrés, et une liste pour modéliser la file des sommets à voir.

```
def ParcoursLargeur(D,s):
    FileAVoir=[s]
    DejaRencontre={}
    DejaRencontre[s]=True
    while len(FileAVoir)>0:
        s2=FileAVoir.pop(0)
        for v in D[s2]:     #D[s2] est la liste des sommets voisins de s2
            if not v in DejaRencontre:
                FileAVoir.append(v)
                DejaRencontre[v]=True
    return DejaRencontre
```

Observons alors qu'il est clair qu'un graphe est **connexe** si et seulement si le parcours en largeur à partir d'un sommet quelconque permet d'atteindre tous les autres sommets. Ainsi, en utilisant la fonction `ParcoursLargeur`, on peut écrire une fonction déterminant si un graphe est connexe ou pas :

```
def test_connexe(D):
    cles=list(D.keys())
    DejaRencontre=ParcoursLargeur(D,cles[0])
    return  len(D)==len(DejaRencontre)
```

4.2 Parcours en profondeur d'un graphe

L'algorithme de parcours en profondeur d'un graphe consiste, partant d'un sommet initial (en argument), à visiter tout le graphe, en passant à chaque fois d'un sommet à l'un de ses voisins non encore visité. Et lorsque l'on arrive à un sommet où tous les voisins ont déjà été visités, on revient en arrière sur le dernier sommet où l'on pouvait suivre un autre chemin, et on prend celui-ci.
Le principe est donc de suivre *jusqu'au bout* tous les chemins du graphe, et à la fin de chaque chemin on revient au dernier embranchement possible.

Pour implémenter cet algorithme, on manipule une pile `PileChemin` contenant tous les sommets visités lors du chemin en cours. Afin d'éviter de visiter un sommet plusieurs fois, il faut de plus garder en mémoire quels sont les sommets déjà visités : on utilise pour cela un dictionnaire `DejaRencontre`.

On implémente l'algorithme comme suit. On commence par mettre le sommet initial dans `PileChemin`, puis tant que `PileChemin` n'est pas vide, on regarde le sommet en haut de la pile :

i. s'il possède un voisin qui n'a pas déjà été visité, alors on empile celui-ci et on le marque comme visité dans `DejaRencontre`;
ii. sinon, on le retire de la pile.

```
def ParcoursProfondeur(D,s):
    DejaRencontre={ }
    PileChemin=[s]
    DejaRencontre[s]=True
    while len(PileChemin)>0:
        s2=PileChemin.pop()
        for v in D[s2] :
            if not v in DejaRencontre:
                PileChemin.append(s2)
                PileChemin.append(v)     #on empile le premier voisin v non déjà
                DejaRencontre[v]=True    #rencontré et on rempile également s2,
                break                    #pour éventuellement étudier le reste
    return DejaRencontre                 #de ses voisins plus tard
```

La complexité du parcours en profondeur d'un graphe est la même que celle du parcours en largeur : en $O(n+p)$, où n est le nombre de sommets et p est le nombre d'arêtes du graphe.

En utilisant un parcours en profondeur, on peut écrire une fonction `CheminProfondeur(D:dict,s1:str,s2:str)->[str]` qui renvoie un chemin menant de $s1$ à $s2$ si un tel chemin existe, et `False` sinon.
Néanmoins ce chemin n'est pas toujours minimal en terme de longueur.

```
def CheminProfondeur(D,s1,s2):
    DejaRencontre={s1:True}
    PileChemin=[s1]
    while PileChemin !=[ ]:
        x=PileChemin.pop()
        for s in D[x] :
            if not s in DejaRencontre:
                PileChemin.append(x)
                PileChemin.append(s)
                DejaRencontre[s]=True
                if s==s2:
                    return PileChemin
                else:
                    break
    return False
```

Remarque 2.

- ✗ Le parcours en profondeur permet de déterminer directement un chemin explicite entre deux sommets : c'est l'avantage du parcours en profondeur par rapport au parcours en largeur.
- ✗ Mais il ne permet pas facilement de déterminer la longueur du plus court chemin entre deux sommets comme le parcours en largeur le fait : c'est l'avantage du parcours en largeur par rapport au parcours en profondeur.
- ✗ Bien entendu, on peut (en s'adaptant convenablement) obtenir un chemin explicite via un parcours en largeur, ou un plus court chemin via un parcours en profondeur, mais c'est moins naturel.

5 Compléments : Pile et File

Parmi les objets itérables, on a aussi les notions de pile et de file implicitement mentionnées précédemment dans les algorithmes de parcours pour les graphes.
On présente d'abord la notion de *pile*, qui est une structure de données plus pauvre que celle de liste, mais qui est donc plus simple à implanter. Les éléments d'une pile (*stack* en anglais) sont organisés de façon linéaire, comme pour une liste. Les opérations possibles sont les suivantes:

- ✗ construire une pile vide;
- ✗ tester si une pile est vide;
- ✗ ajouter (on dira *empiler*) un élément au sommet de la pile (opération appelée *push* en anglais);
- ✗ supprimer (on dira *dépiler*) l'élément au sommet de la pile (opération appelée *pop* en anglais), et ainsi accéder à cet élément.

Cette structure correspond à l'image d'une pile de cartes ou d'assiettes posées sur une table: on ne peut pas accéder à un élément au fond de la pile, mais seulement à celui se situant au sommet (qui est le dernier élément ajouté).

Un exemple d'application est la suite des appels récursifs d'une fonction, qui est stockée en mémoire à l'aide d'une pile. Plus généralement, les processeurs génèrent une pile de façon native pour gérer les appels et retours de sous-programme. Citons encore l'historique des pages webs visitées, ou la commande d'annulation des frappes au clavier, qui se comportent comme s'ils utilisaient des piles.
Il existe également une structure de données dans laquelle c'est le premier élément arrivé qui sera le premier à ressortir (FIFO, pour *first in, first out* en anglais). On parle alors de structure de *file* (*queue* en anglais), par analogie avec les files d'attentes se formant aux caisses d'un magasin.