Dictionnaires

I. Préliminaires

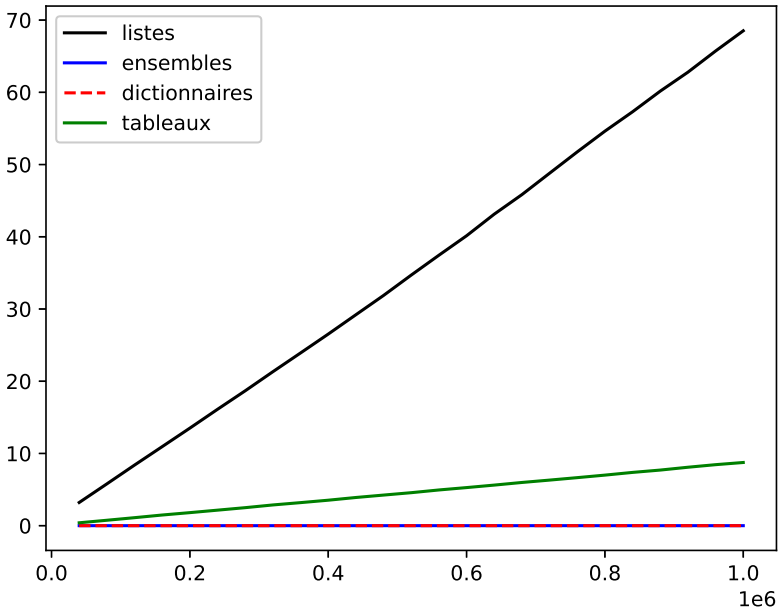
Une des problématiques importantes de l'informatique est le stockage des données. Pour traiter efficacement ces dernières, il faut les ranger de manière adéquate, en fonction du problème posé. L'objet informatique qui stocke des valeurs en mémoire s'appelle une *structure de données* ; elle est caractérisée par les opérations qu'elle permet et le coût de ces opérations.
Les principales structures de données en Python sont : les *listes*, les *ensembles*, les *dictionnaires*, et les *tableaux* `numpy`.

Chacune de ces structures a des avantages et des inconvénients, qui dépendent bien sûr du problème posé.

Le petit programme Python ci-dessous compare les temps mis pour tester si un élément appartient à la structure S. Pour cela on a créé une structure formé de n élémens, chacun de ses éléments étant un entier choisi au hasard entre 0 et N. Puis pour chaque entier $k \in [\![0\,;2N]\!]$, on compare le temps des instructions `k in S`.

```
import numpy as np
import matplotlib.pyplot as plt
import random
import time

n = 1000001 # taille maxi des tableaux
pas = 40000 # écart entre les tailles de deux tableaux d'essais
N = 10000 #  max des valeurs

essais = list(range(pas, n, pas))
Temps_listes = [0]*len(essais)
Temps_ensembles = [0]*len(essais)
Temps_dicos = [0]*len(essais)
Temps_tableaux = [0]*len(essais)

for i, n in enumerate(essais):
    print(n)
    # les 4 structures:
    liste = []
    dico = {}
    for k in range(n):
        x = random.randint(0,N)
        liste.append(x)
        dico[x] = 0
    ensemble = set(liste)
    tableau = np.array(liste)

    # test des temps
    deb = time.perf_counter()
    for k in range(2*N):
        k in liste
    Temps_listes[i] = time.perf_counter() - deb

    deb = time.perf_counter()
    for k in range(2*N):
        k in ensemble
    Temps_ensembles[i] = time.perf_counter() - deb

    deb = time.perf_counter()
    for k in range(2*N):
        k in dico
    Temps_dicos[i] = time.perf_counter() - deb
```

```
        deb = time.perf_counter()
        for k in range(2*N):
            k in tableau
        Temps_tableaux[i] = time.perf_counter() - deb

    plt.plot(essais, Temps_listes, color='black', label = 'listes')
    plt.plot(essais, Temps_ensembles, color='blue', label = 'ensembles')
    plt.plot(essais, Temps_dicos, '--', color='red', label = 'dictionnaires')
    plt.plot(essais, Temps_tableaux, color='green', label = 'tableaux')
    plt.legend(loc="upper left")
    plt.show()
```



On voit bien sur ce graphique que le temps d'accès à un élément dans une liste ou un tableau est de complexité temporelle linéaire $O(n)$ (avec un gros avantage pour les tableaux `numpy`), alors que celle pour les ensembles ou les dictionnaires est de $O(1)$ (temps constant).

Si l'on veut pouvoir accéder rapidement à un élément dans une structure de données, il faut donc choisir une structure de type `dictionnaire` (les ensembles n'étant en fait qu'un cas simplifié de dictionnaire, où il n'y a que les clés mais pas de valeurs). La raison de ces bonnes performances seront expliquées dans une prochaine section.

II. Les dictionnaires

Déf 1:

Soient deux ensembles C (ensemble de clefs) et V (ensemble de valeurs).
Un dictionnaire (ou mieux, un tableau associatif) construit sur C et V est une fonction de C dans V, c'est-à-dire un sous-ensemble T de $C \times V$ tel que, pour tout élément $c \in C$ il existe *au plus* un élément $v \in V$ tel que (c, v) appartienne à T (cette fonction s'appelle une *association*).

De plus, on exige qu'un dictionnaire supporte les opérations suivantes (dont on donne la syntaxe en Python) :

- *Création d'un dictionnaire vide* : `d={}` ou `d=dict()`. On peut aussi créer un dictionnaire en énumérant son contenu par : `d = {c1:v1, c2:v2, ..., cn:vn}` où les c_i sont les clés et les v_i les valeurs..
- *Insertion d'un couple clé-valeur* (c, v) : `d[c] = v`. Cette instruction sert aussi à modifier une association existante.
- *Suppression d'une clé (et de la valeur associée)* : `del d[c]`.
- *Recherche d'une clé en temps* $O(1)$: `c in d` (booléen).

De même que les éléments d'une liste peuvent être des listes, les éléments d'un dictionnaire peuvent être des dictionnaires. Exemple :

```
>>> tel = {'Abel': {'fixe': '012345678', 'mobile': '06123456'},
...         'Gauss': {'mobile': '06666689', 'fixe': '02478956'},
...         'Cauchy': {'fixe': '03256789', 'mobile': None}
... }

>>> print(tel['Cauchy']['fixe'])
03256789
```

On peut accéder aux éléments d'un dictionnaire à l'aide des méthodes `keys` et `values`. La méthode `items` donne accès aux couples (clé, valeur).

```
tel = {'Abel': '0123456', 'Gauss': '0645789', 'Cauchy': None}

print(list(tel.keys()), list(tel.values()),'\n')

for couple in tel.items():
    print(couple)
```

```
['Abel', 'Gauss', 'Cauchy'] ['0123456', '0645789', None]

('Abel', '0123456')
('Gauss', '0645789')
('Cauchy', None)
```

D'autres fonctions faciles à comprendre :

```
>>> tel = {'Abel': '0123456', 'Gauss': '0645789', 'Cauchy': None}

>>> 'Pythagore' in tel, 'Gauss' in tel
(False, True)

>>> len(tel)
3

>>> del(tel['Cauchy'])

>>> len(tel)
2
```

Comme les listes, les dictionnaires sont des objets mutables, il est donc préférable d'utiliser la méthode `copy` pour copier un dictionnaire.

```
>>> tel = {'Abel': '0123456', 'Gauss': '0645789', 'Cauchy': None}

>>> tel2 = tel

>>> tel2['Abel'] = None

>>> tel, tel2
({'Abel': None, 'Gauss': '0645789', 'Cauchy': None}, {'Abel': None, 'Gauss': '0645789', 'Cauchy': N

>>> tel3 = tel.copy()

>>> tel3['Abel'] = '0123456'

>>> tel, tel3
({'Abel': None, 'Gauss': '0645789', 'Cauchy': None}, {'Abel': '0123456', 'Gauss': '0645789', 'Cauch
```

III. Implémentation d'un dictionnaire

L'implémentation efficace d'un dictionnaire repose sur l'utilisation de tables de hachage.

Notons $\mathcal{U}$ l'ensemble de toutes les clés possibles.
Si $\mathcal{U}$ était de la forme $[\![0\,;m-1]\!]$ avec $m \in \mathbb{N}^*$, on pourrait directement stocker les éléments du dictionnaire dans un tableau de taille m. Mais cela n'est pas envisageable, d'une part parce que l'univers de toutes les clés possibles est très grand, d'autre part parce que la recherche d'un élément dans un tableau de taille m est en $O(m)$ alors qu'on veut un $O(1)$.

La solution consiste à utiliser une fonction de hachage $h : \mathcal{U} \to [\![0\,;m-1]\!]$ qui à toute clé c associe un élément $h(c)$ de $[\![0\,;m-1]\!]$ et un tableau T de taille m pour stocker les couples (clé, valeur). Cependant, le cardinal de $\mathcal{U}$ est bien plus grand que m, donc la fonction h ne peut être injective : deux éléments de clés différentes peuvent se retrouver au même emplacement du tableau T : on parle de collision. Dans ce cas, si par exemple $h(c_1) = h(c_2) = k$, on stockera dans $T[k]$ la liste formée des couples (c_1, v_1) et (c_2, v_2) (où v_i est la valeur de la clé c_i. Cette liste s'appelle un paquet.

La taille moyenne de chaque paquet est le facteur de remplissage $\alpha = \dfrac{n}{m}$, où n est le nombre de clés utilisées. Si α est suffisamment petit, la recherche d'un élément dans chaque paquet sera rapide. Pour cela, la fonction h doit être choisie de manière à ce qu'elle prenne des valeurs les plus uniformes possibles dans $[\![0\,;m-1]\!]$.

Construction de la fonction de hachage

Pour construire h, il faut d'abord construire une fonction $f : \mathcal{U} \to \mathbb{N}$ qui à toute clé possible associe un entier. Lorsque les clés sont des chaînes de caractère, une fonction possible est proposée dans l'exercice suivant.

EXERCICE 1

Cet exercice utilise pour simplifier le codage ASCII sur 8 bits des caractères : chaque caractère c est codé par un entier entre 0 et 255, par l'appel à la fonction Python `ord(c)` (réciproquement, l'appel à `chr(k)` donne le caractère de code k).

Écrire une fonction `chaine_vers_entier` qui à une chaîne `ch` associe l'entier

$$\sum_{k=0}^{\ell-1} \text{ord(ch[k])} 256^k$$

où ℓ est la longueur de la chaîne. On utilisera la méthode de Hörner pour réduire la complexité.

La fonction f étant ainsi construite, il y a ensuite plusieurs façons de construire la fonction de hachage h.

- *Méthode de la division*

  Il s'agit de la méthode la plus simple. Pour une table de hachage de taille m, on pose

  $$\forall\, c \in \mathcal{U},\ h(c) = f(c) \bmod m$$

  (ce qui s'écrit en Python : `f(c)%m`).
  La qualité de ce hachage dépend fortement des valeurs de m. Dans la pratique, on choisit pour m un nombre premier éloigné des puissances de 2.

- *Méthode de la multiplication*

  On se donne un réel $\theta \in]0\,;1[$. Un bon choix pour θ est une valeur proche de $\frac{\sqrt{5}-1}{2}$. Puis pour tout entier c on pose :

  $$hc = \lfloor m(c\theta - \lfloor c\theta \rfloor) \rfloor\,.$$

  On obtient bien ainsi un entier de l'intervalle $[\![0\,;m-1]\!]$ puisque $c\theta - \lfloor c\theta \rfloor$ appartient à $[0\,;1[$.

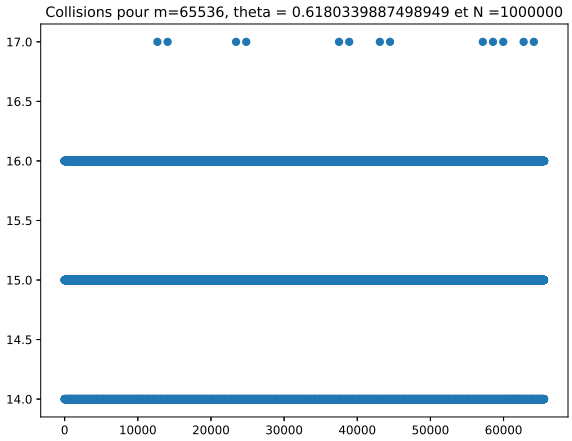
  Remarques

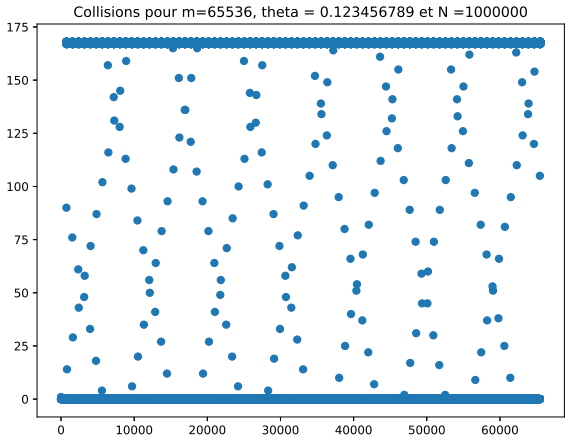
  1. Les calculs ayant lieu en base 2, un bon choix pour m est une puissance de 2, et pour θ un nombre de la forme $\frac{x}{2^p}$ proche de $\frac{\sqrt{5}-1}{2}$.
  2. Il est facile de vérifier que si $\theta = \frac{p}{q} \in \mathbb{Q}$ alors h prend au plus $\min(m, q)$ valeurs. Il ne faut donc pas choisir pour θ un nombre rationnel avec un dénominateur petit !

EXERCICE 2

Écrite une fonction `test_hachage` qui étant donnés m et θ, calcule pour chaque entier $k \in [\![0\,;m-1]\!]$ le nombre de clés $c \in [\![0\,;N]\!]$ (N entier donné, grand) tels que $h(c) = k$ (nombre de collisions), et qui représente graphiquement ce nombre de collisions.

On obtient par exemple les résultats suivants :





- Notons que Python dispose d'une fonction de hachage, `hash`, qui s'applique à des objets de type *non mutables* (entiers, flottants,chaines, tuples....); c'est d'ailleurs pour cette raison que les clés d'un dictionnaire doivent être d'un type non mutable.
  Voici des exemples :

```
>>> hash(123456789)
123456789

>>> hash('123456789')
4075558362725385177

>>> hash( (1,2,3,4,5,6,7,8,9) )
2885636760529661641

>>> hash( [1,2,3,4,5,6,7,8,9] )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```