Leçon d'informatique : Recherche séquentielle – dictionnaire – tuple

S. Benlhajlahsen - PCSI$_1$

Sommaire

Introduction : Dans cette leçon, on commencera par introduire deux nouveaux objets structurés : les ***tuple*** et les **dictionnaire**.
On se penchera ensuite sur la recherche d"un élément dans un ensemble par recherche séquentielle ou recherche linéaire, c'est-à-dire une recherche naïve en balayant les éléments de l"ensemble.

I Tuple

À retenir : Un *tuple*[a] est un ensemble **indexé**[b] délimité par des parenthèses. Les éléments du `tuple` peuvent être de type différents. Par exemple, `a = (10,20,30,1.52,40,"mama", False)` est un *tuple* à 7 éléments composés d'entiers, de flottants et de booléen.

a. l'équivalent d'un uplet en mathématique.
b. ou ordonné

- `()` est le *tuple* vide et `(3,)` est un *tuple* à un élément[1] ;
- `len( (1,3,5) )` renvoie la « longueur » du *tuple*, c'est-à-dire son nombre d"éléments. Ainsi, `len((1,3,5))` renvoie 3 et, pour `a = (10,20,30,1.52,40,"mama", False)`, `len(a)` renvoie 7.
- On accède aux éléments par leurs indices. Ainsi, `(1,3,5)[1]` renvoie `3` et `a[5]` renvoie `"mama"`.
- l"opérateur `+` permet la concaténation. Ainsi, `(1,2)+(10,20)` renvoie `(1,2,10,20)` ;
- L"opérateur `*` permet la répétition. Ainsi, `3*(10,20)` renvoie `(10, 20, 10, 20, 10, 20)`.
- On peut hacher/slicer un *tuple*. Ainsi, `(10,20,30,1.52,40,"mama", False)[1:4]` renvoie les éléments compris entre l'indice 1 et l'indice 4 exclu soit encore `(20, 30, 1.52)`.

Comparaison avec la liste : mutabilité ou immutabilité ? À ce stade, il n'y a pas de différence avec une liste. Cependant, on dira que :

- une liste est **mutable** ou **muable** car on peut modifier les éléments de la liste sans nécessairement construire une nouvelle liste.
- un tuple est **immuable** ou **immutable**. On ne peut pas modifier les éléments du tuple. On doit alors construire un nouveau *tuple*.

Mot-clé `tuple` Comme pour les mots-clé `list` ou `str`, c'est à la fois le type de l'objet et une fonction qui convertit. Ainsi :

Test de type	`type([1,2]) == list`	`type("12") == str`	`type((1,2)) == tuple`	`type({"pomme":1}) == dict`
renvoie	`True`	`True`	`True`	`True`

Remarque : la fonction suivante

```
def f(x, y):
    return x**2, y**2
```

renvoie naturellement un *tuple*.

1. Attention, `(3)` est considéré comme l"entier 3.

```
>>> L = [13,10,20]
>>> L[0]
13
>>> L[0] = 100
>>> L
[100, 10, 20]
```

```
>>> T = (13,10,20)
>>> T[0]
13
>>> T[0] = 100
TypeError: 'tuple' object does not support
    item assignment
```

FIGURE 1 – Mutabilité de la liste et immutabilité du *tuple*.

II Dictionnaire

Remarque Les tuples, listes et chaînes partagent une même structure d'indexation : leurs éléments sont indexés entre 0 et `n-1` si n est la longueur de l'objet. À présent, on va introduire les dictionnaires dont les éléments ne sont pas des entiers.

À retenir : Un dictionnaire est un ensemble d'éléments, délimité par des accolades . Les éléments sont indexés par des **clés** qui peuvent être des entiers, des flottants, des chaînes…

- `{}` est le dictionnaire vide ;
- On considère l'exemple de la figure 2. Supposons qu'on ait un panier de fruits composés de 5 pommes, 20 bananes et 10 fraises. On construit un dictionnaire qui associe le fruit à la quantité. Dans le programme, le dictionnaire peut être créer en une seule étape ou progressivement. On dit que `"pomme"`, `"banane"` et `"fraise"` constituent les **clés** du dictionnaire[2] et `5`, `20` et `10` constituent les **valeurs** du dictionnaire[3].
- pour un dictionnaire `d`, on accède à ses valeurs par `d.values()` ;
- pour un dictionnaire `d`, on accède à ses clés par `d.keys()` ;

  Remarque : Si on veut la liste des clés, on peut utiliser la synthaxe `list(d.keys())`. De même, le *tuple* des clés sera obtenu par `tuple(d.keys())`.

```
# on construit le dico en une seule étape
d = {"pomme":5, "banane":20, "fraise":10}
type(d)            # affiche <class 'dict'>
d.keys()           # affiche les clés
d.values()         # affiche les valeurs
d['pomme']         # affiche 5
len(d)             # affiche 3
```

```
d = {}             # dictionnaire vide
#        on le remplit progressivement
d["pomme"] = 5
d["banane"] = 20
d["fraise"] = 10
print(d) # affiche
# {'pomme':5,'banane':20,'fraise':10}
```

FIGURE 2 – Exemple de création de dictionnaire.

- `len(d)` permet d'accéder la longueur du dictionnaire.

  Remarque : Il ne peut pas y avoir deux clés identiques. Par contre, deux valeurs peuvent être identiques[a]. Ainsi, `len(d)` renvoie le nombre de clés du dictionnaire.

  a. Par exemple, `{"a":1,"b":1}` est un dictionnaire pour lequel les deux clés sont associés à la même valeur.

Comparaison des objets structurés : La table 1 récapitule les caractéristiques des principaux objets structurés.

2. *keys* en anglais.
3. *values* en anglais.

objet	liste	chaîne de caractère	tuple	dictionnaire
délimiteurs	crochets	guillemets ou apostrophes	parenthèses	accolades
objet vide	`[]`	`""` ou ’’	`()`	`{}`
type associé	`list`	`str`	`tuple`	`dict`
Exemple	`[1,2]`	`"Adresse123"`	`(1,2)`	`{"pomme":1,"banane":2}`
Muable ?	oui	non	non	oui
Hachage/*slicing* ?	oui	oui	oui	non
Ordonné ?	oui. `[1,2]==[2,1]` renvoie `False`	oui. `"12"=="21"` renvoie `False`	oui. `(1,2)==(2,1)` renvoie `False`	non. `{"a":1, "b":2} == {"b":2, "a":1}` renvoie `True`

TABLE 1

III Recherche séquentielle

À retenir : Une recherche **séquentielle** ou **linéaire** est une recherche sur un objet structuré[a] en balayant les éléments de la liste.

a. liste, tuple, dictionnaire, chaîne

III.A Recherche d’un élément

Recherche d’un élément dans une liste On peut chercher l’apparition d’un élément `a` dans une liste `L` (voir figure 3). La fonction `recheche1` balaie donc l’ensemble des éléments et renvoie `True` si `a` est présent.
Cependant, on constate que, si `a` est parmi les premiers éléments, balayer le reste de la liste est inutile. On peut alors préférer la seconde méthode car `recherche2` propose de « casser » la boucle `for`[a].

a. comme avec l’instruction `break`

```python
# recherche de a dans L
def recherche(a, L):
    res = False
    for x in L :
        if x==a:
            res = True
    return res
print(recherche(1, [4,3,1,2]))
```

```python
def recherche(a, L):
    for x in L :
        if x==a:
            return True
    return False
print(recherche(1, [4,3,1,2]))
```

FIGURE 3 – Recherche d’un élément dans une liste.

Autre solution : utilisation de `in` La synthaxe `a in L` renvoie directement `True` ou `False` suivant l’occurence ou non de `a` dans `L`.

III.B Recherche du maximum

Recherche du maximum : Un autre exemple de recherche linéaire est la recherche du maximum d’une liste ou d’un *tuple* composé d’entier ou de flottants (voir figure 4). On peut aussi chercher le maximum des valeurs d’un dictionnaire. On renvoie à chaque fois la position (ou la clé) et la valeur du maximum

```python
# recherche du maximum d'une liste ou d'un tuple
def maximum(L):
    res = L[0]
    p = 0
    n = len(L)
    for i in range(1, n):
        if L[i]>res:
            p = i
            res = L[i]
    return p, res
print(maximum( [10, 30, 20]) ) # liste
print(maximum( (10, 30, 20)) ) # tuple
```

```python
# recherche du maximum d'un dictionnaire de valeurs entières
def maximum(dico):
    # on recupère la "première" clé
    p = list(dico.keys())[0]
    res = dico[p]
    for cle in dico:
        if dico[cle]>res:
            p = cle
            res = dico[cle]
    return p, res
print(maximum({"a":1, "b":4, "c":2}))
```

FIGURE 4 – Recherche du maximum d'une liste, d'un tuple ou des éléments d'un dictionnaire.

III.C Comptage des éléments à l'aide d'un dictionnaire

À ce stade, les dictionnaires ne présentent pas un gros intérêt. Supposons qu'on veuille énumérer les occurences des lettres `"a"`, `"b"`, `"c"` et `"d"` dans une chaîne de caractère `ch`. On propose une solution en figure 5. La solution de gauche prend en argument une chaîne qui ne sont composées que de ces quatre lettres et la deuxième solution fonctionne avec une chaîne quelconque.

```python
def apparition(ch):
    # ch est une chaine composé uniquement des quatres lettres
    # minuscules "a", "b", "c" et "d"
    dico = {"a": 0,"b":0, "c":0, "d":0}
    for x in ch:
        dico[x] +=1
    return dico
print(apparition("abcdaaad"))
```

```python
def apparition2(ch):
    # ch est une chaine quelconque
    dico = {"a": 0,"b":0, "c":0, "d":0}
    for x in ch:
        if x in dico.keys():
            dico[x] +=1
    return dico
print(apparition2("abcdefgaaa"))
```

FIGURE 5 – Dictionnaire d'occurence des lettres d'une chaîne.

III.D Coût de la recherche linéaire

Complexité temporelle : On appelle **complexité temporelle** une mesure du nombre d'opérations élémentaires qu'effectue un programme.
Les opérations élémentaires sont, par exemple, les additions, multiplications, affectations, comparaisons pour lesquelles les processeurs sont spécialement fabriqués.
Enfin, la mesure du nombre d'opération pourra être rapprochée du temps que mettra un programme.

Exemple de la recherche linéaire : Si on prend l'exemple de `recherche1` de la figure 3, on note n la taille de la liste. On constate que l'on fait :

- une affectation `res = False`;
- à chaque passage dans la boucle, on fait :
  - l'extraction de l'élément `x`;
  - une comparaison `if x==a`;
- une éventuelle affection de `True` à `res`.

Cela donne un nombre d'opération qui est de l'ordre de $T_n = 3n + 1$ voir $3n + 2$ dans le pire des cas.
Si n devient très grand, le nombre d'opération est de l'ordre de $T_n \approx 3n$. On constate alors que la croissance du nombre d'opération est **linéaire**. On dira que la **complexité est linéaire en** $O(n)$.

Notation en « grand O » : Soient f et g deux fonctions de la variable réelle x. On dit que f est dominée par g si :

$$\exists (A, C) \in \mathbb{R}_+^2, \forall x > A, \ |f(x)| \leq C\,|g(x)|$$

et on note $f(x) = O(g(x))$. Intuitivement, cela signifie que f ne croît pas plus vite que g.

Mesure du temps d'exécution : Cette complexité linéaire doit pouvoir se traduire par un temps d'exécution qui croît linéairement. On se propose donc d'utiliser la méthode `time()` du module `time` qui permet de mesurer un instant t. On génère alors aléatoirement des listes d'entiers de taille n et on cherche un élément `a`. On mesure alors le temps d'exécution de la recherche linéaire. En trait plein, on trace la courbe correspondant à la fonction `recherche1` et en tirets la courbe pour `recherche2`.

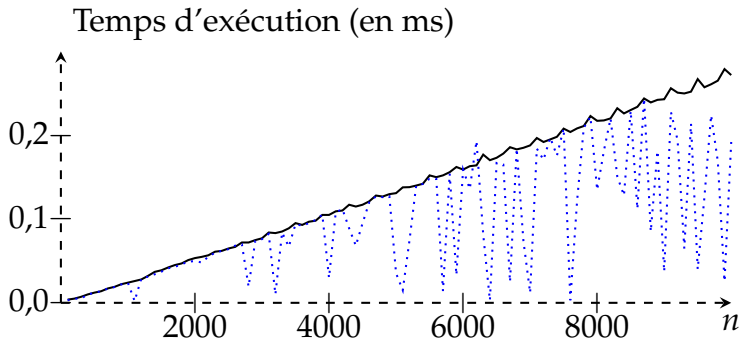

FIGURE 6

```
from random import sample, randint
from time import time
def recherche1(a, L):
    res = False
    for x in L :
        if x==a:
            res = True
    return res
def recherche2(a, L):
    for x in L :
        if x==a:
            return True
    return False
N = 10000
L0 = []
L1 = []
L2 = []
for n in range(10, N+1, N//100):
    L0.append(n)
    L = sample(range(N+1), n)
    a = randint(0,n)
    t1 = time()
    recherche1(a, L)
    t2= time()
    L1.append(t2-t1)
#########
    t1 = time()
    recherche2(a, L)
    t2= time()
    L2.append(t2-t1)
```

FIGURE 7 – Etude du temps d'exécution de la recherche linéaire.