complet

January 19, 2021

Algorithmique des tableaux

Christophe Saint-Jean

Transparents du cours

Code du cours

Année 2020-2021

1 Organisation de l'UE

1.1 Mentions Légales

1.2 L'équipe enseignante

- **Christophe Saint-Jean** (CM/TD/TP - Resp.)
- Laurent Mascarilla (TD/TP)
- El Hadi Zahzah (TD/TP)

1.3 Communication

- Questions pédagogiques : Moodle
  - Appronfondissement/Questions
  - Organisation de l'UE/Planning
- Questions administratives (Secrétariat)
  - Appartenance groupes TD/TP
  - Absences/Justifications

1.4 Dispositif horaire

On essaie tout en **synchrone**:

- 6 cours de 1,5 heures
- 4 TDs de 1,5 heures
- 5 TPs de 1,5 heures
- 4 créneaux de 1,5h de TEA - 2 sujets

1.5 Evaluation

$$S_1 = \frac{CC_1 + CC_2}{2}$$

$$S_2 = CC_3$$

Les CC se passent *a priori* en TP 3 et 5 (idem semestre 1).

Attention à la règle sur les absences.

1.6 Environnement de travail

- Cours:
  - En direct: notebook
  - Hors-ligne: pdf, html (parfois des pbs de conversion).
- TD:
  - Support papier
  - Eventuellement un IDE (Thonny)
- TP/TEA:
  - Un IDE (thonny)

1.7 Les objectifs de cet enseignement

- Initiation à l'algorithmique
  - Qu'est ce qu'un algorithme ?
  - Différence algorithme / programme (cf. TD)
  - Initiation à l'analyse d'un algorithme.
- Approfondir vos connaissances sur :
  - les listes, tableaux, fonctions.
  - le langage Python.
- Découvrir des algorithmes simples et les analyser.
- Initiation à la récursivité.

2 Généralités sur l'algorithmique

2.1 Analogie algorithme/recette

1. Mettez la farine dans un saladier avec le sel et le sucre.
2. Faites un puits au milieu et versez-y les œufs légèrement battus à la fourchette.
3. Commencez à incorporer doucement la farine avec une cuillère en bois. Quand le mélange devient épais, ajoutez le lait froid petit à petit.
4. Quand tout le lait est mélangé, la pâte doit être assez fluide, si elle vous paraît trop épaisse, rajoutez un peu de lait. Ajoutez ensuite le beurre fondu, mélangez bien.
5. Faites cuire les crêpes dans une poêle chaude.
6. Répétez jusqu'à épuisement de la pâte.

On attend:

- Langage de description intelligible

- Instructions séquentielles, fonctions, répétitives.

On distingue bien:

- la recette (l'algorithme)
- de sa mise en oeuvre (traduction dans un langage de programmation =: Implémentation)
- aux moyens d'ustentiles de cuisine (variables, structures de données, fonctions, …)

2.2 Algorithme (Définition)

Une définition formelle:

```
Un algorithme est la description d'une méthode de calcul qui, à partir d'un
ensemble de données d'entrée (problème) et une suite finie d'étapes, produit un
ensemble de données en sortie (solution).
```

```
Un algorithme est la description d'une méthode de calcul ...
```

Description

On doit décrire chaque fonction (sous-algorithmes) non triviale et structures de données employées

Méthode de calcul

Il ne peut résoudre que des problèmes calculables. On démontre que certains problèmes ne sont pas calculables (décidables).

Ex.: Problème de l'arrêt (-> L3)

```
Un algorithme est .... et une suite finie d'étapes, produit un ensemble de
données en sortie (solution).
```

Une suite finie d'étapes

Attention, ne pas confondre:

- Une séquence d'instructions qui se termine.
- Une description de longueur finie possibilité de boucle infinie.

Solution

L'algorithme apporte t'il une solution au problème posé ?

2.3 Algorithmique (Définition)

```
L'algorithmique est la science qui étudie les algorithmes pour eux-même
indépendamment de tout langage de programmation.
```

d'après "Al Khwarizmi", surnom du mathématicien arabe Muhammad Ibn Musa (IX siècle).

2.3.1 Algorithmique (Histoire de calculs)

L'algorithmique et les algorithmes sont bien antérieurs à l'informatique:

- Abaques grecques, romaines, chinoises (arithmétique simple, racines carrées)
- PGCD d'Euclide, Crible d'Ératosthène (IIIème siècle av. J.-C.)
- Boulier japonais (XIIIème siècle)

- Pascaline (1646)
- Métier à tisser programmable Jacquard (18ième siècle)
- Algorithme de cryptographie, Traitement des données, Algorithmique optique/quantique

2.3.2 Questions de l'algorithmique

1. L'algorithme A est t'il correct pour un problème P (toutes instances) ?
2. L'algorithme A se termine t'il ?
3. L'algorithme A est t'il plus efficace qu'un algorithme B ?

Parallèlement, des questions plus fondamentales:

- Est il possible de trouver un algorithme qui résoud un problème P ? (Décidabilité)
- Si, oui existe t'il un algorithme efficace résoud un problème P ? (Classes de complexité)

2.3.3 Exemples d'algorithmes (cf. S1)

Algorithme 1: Plus grand élément d'un tableau

Données : T un tableau n de valeurs comparables par $<$
Résultat : La plus grande valeur de T

```
1 max ← T[1];
2 pour i ← 2 à n faire
3 |   si max < T[i] alors
4 |   |   max ← T[i];
5 |   fin
6 fin
7 retourner max
```

Algorithme 2: Approximation de π

Données : π_c une valeur approchée précise de π, p un entier positif
Résultat : Une approximation $\hat{\pi}$ de π_c à 10^{-p} près

```
1 som ← 0 ;                          // Initialiser som à 0
2 π̂ ← 0 ;                            // Initialiser π̂ à 0
3 tant que |π̂ − π_c| ≥ 10^{-p} faire
4 |   som ← som + (−3^k)/(2k+1) ;     // Ajouter (−3^k)/(2k+1) à som
5 |   π̂ ← √12 som;
6 |   k ← k + 1 ;                     // Incrémenter k
7 fin
8 retourner π̂
```

2.4 Description d'un algorithme

On utilisera un langage de description d'un algorithme appelé **pseudo-code**.

Caractérisques du pseudo-code

- Il ne doit pas être attaché la syntaxe d'un langage informatique particulier.
- Il doit être lisible par un non-programmeur.
- Être capable de décrire les structures de contrôle des langages impératifs (If, While, For, …).

2.4.1 L'interface de l'algorithme

Quelques sont les entrées attendues par l'algorithme ?

- Type : Nombre, Tableau, Liste, Arbre, etc …
- Taille : nombre de bits, nombre d'éléments du tableau, nombre de feuilles, …
- Propriétés : entiers positifs, tableau trié, …

Que fait/produit l'algorithme ?

- *Idem* que sur les entrées
- Description textuelle (éventuelle) de l'algorithme

2.4.2 Les types utilisables

Types de données élémentaires :

- Variables simples : booléen, entier, réel, caractère
- Tableaux
- *Pointeurs*

Cela permet de définir des structures de plus haut niveau:

- Chaîne de caractères
- Ensemble, Collection
- Liste, *table de hachage*
- *Graphes, …*

2.4.3 Quelques instructions du pseudo-code

- Affectation : <-
- Test : =
- Opérations arithmétiques : +,-,*,/
- Séparateur d'instructions : " ;"
- Elements d'un tableau T : "T[i]" (convention 1..n !!)
- Adresse d'une variable "@"
- Instruction de retour : Retourner <val. sortie>
- Le branchement conditionnel :

```
Si <condition> alors
  <blocsi>
sinon
  <blocsinon>
```

```
fin
```

- Les itératives et les répétitives:

```
Pour i <- 1 à n [par pas de 1] faire
    <bloc>
fin

Tant que <condition> faire
    <bloc>
fin
```

2.4.4 Langage de description et Python

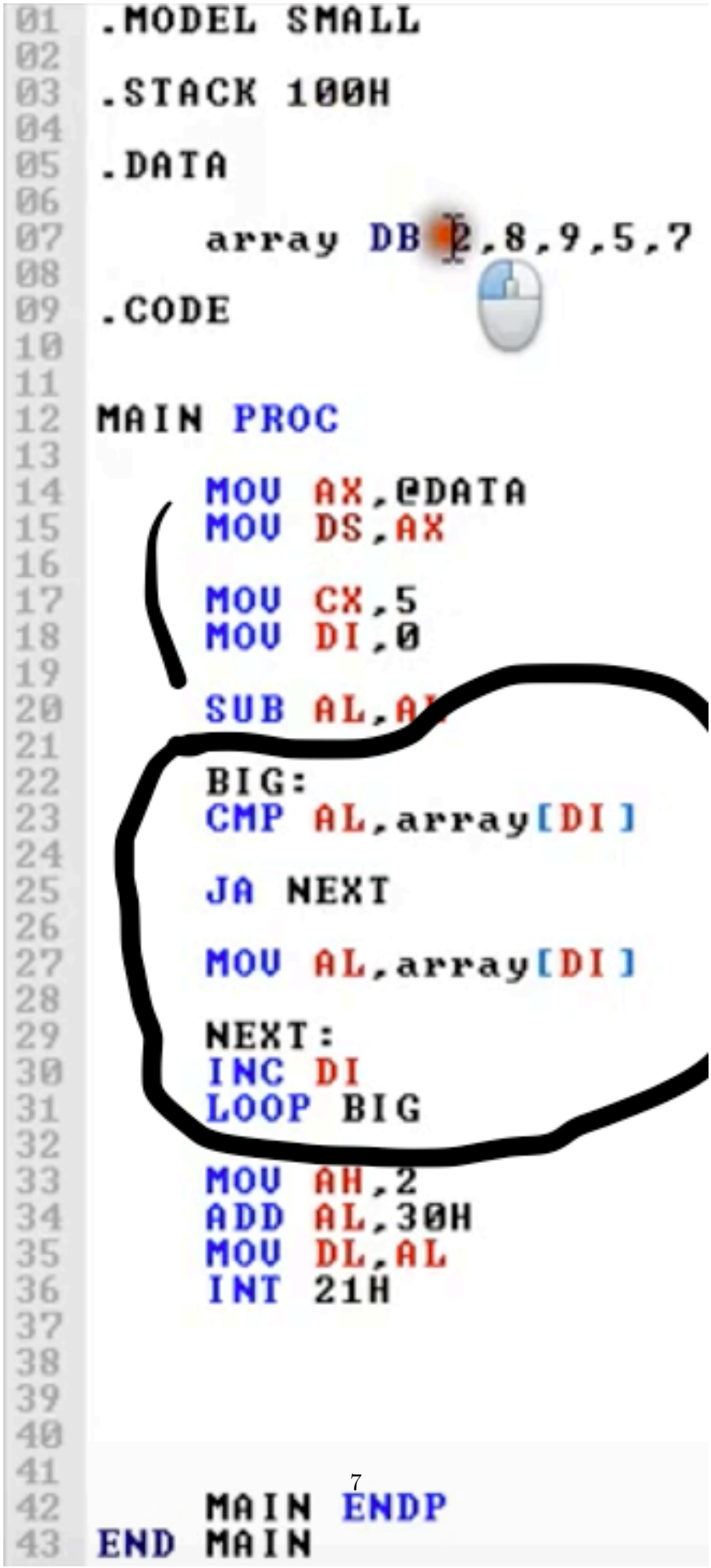
- Le langage Python a été conçu pour être le plus lisible et naturel possible.
- On est très proche du langage de description (raccourci en TD)
- **Un algorithme devient une fonction !!**

```python
def max(T):
    maxi = T[0]
    for Ti in T[1:]:
        if maxi < Ti:
            maxi = Ti
    return maxi
```

En C++

```cpp
template<class T> T max(const T* data, int size) {
    T result = data[0];
    for(int i = 1 ; i < size ; i++)
      if(result < data[i])
        result = data[i];
    return result;
  }
```

En Assembleur



```
.MODEL SMALL

.STACK 100H

.DATA

    array DB 2,8,9,5,7

.CODE


MAIN PROC

    MOV AX,@DATA
    MOV DS,AX

    MOV CX,5
    MOV DI,0

    SUB AL,AL

    BIG:
    CMP AL,array[DI]

    JA NEXT

    MOV AL,array[DI]

    NEXT:
    INC DI
    LOOP BIG

    MOV AH,2
    ADD AL,30H
    MOV DL,AL
    INT 21H



    MAIN ENDP
END MAIN
```

2.4.5 De l'importance d'une bonne description

Algorithme 3: Calcul de l'élément maximal d'un tableau trié

Données : T un tableau trié dans l'ordre croissant de n entiers
Résultat : $i_{\max}$ et max tel que $T[i_{\max}] = \max$ avec max comme la plus grande valeur de T

```
1 i_max ← n;
2 max ← T[n];
```

2.5 Analyse: Preuve de terminaison

- Vérifier que chaque instruction simple se termine:
  - calcul simple, affectation OK
  - affichage OK
  - Appel de fonction -> vérifier la fonction
- Instruction Si : vérifier la condition et les deux branches possibles
- Pour les boucles for, s'assurer que la séquence parcourue est taille finie.
- Pour la répétitive While, s'assurer que dans tous les cas que la condition de continuation sera fausse au moins une fois (critère math. parfois).
- Pour les algorithmes récursifs (plus tard), on doit s'assurer que la récursion se termine

Algorithme 1: Plus grand élément d'un tableau

Données : T un tableau n de valeurs comparables par $<$
Résultat : La plus grande valeur de T

```
1 max ← T[1];
2 pour i ← 2 à n faire
3 |   si max < T[i] alors
4 |   |   max ← T[i];
5 |   fin
6 fin
7 retourner max
```

2.6 Analyse: Preuve de correction

Il est question de prouver que l'algorithme fait ce qu'il dit faire !

On utilise souvent un invariant de boucle et la preuve par récurrence.

Pour des algorithmes concernant un tableau T[1..n], la démarche (simplifiée) est généralement la suivante:

- Soit i l'indice du parcours de T (c.a.d. T[i])
- Avant l'itération, quelle propriété vérifie la variable qui sera retournée par rapport à T[1..i-1] ?

- Après l'itération, cette propriété reste t'elle vraie pour T[1..i] ?
- L'initialisation de la variable est elle compatible à la propriété ?

On a donc les deux éléments d'une récurrence: Initialisation et Hérédité.

Algorithme 1: Plus grand élément d'un tableau

Données : T un tableau n de valeurs comparables par $<$
Résultat : La plus grande valeur de T

```
1 max ← T[1];
2 pour i ← 2 à n faire
3 |   si max < T[i] alors
4 |   |   max ← T[i];
5 |   fin
6 fin
7 retourner max
```

2.7 Analyse : Complexité algorithmique (asymptotique)

L'algorithme est il rapide ?

Pour un tableau T de taille n, la rapidité *devrait* dépendre de:

- n la taille T (des données).
- d'une propriété, du contenu de T.
- du langage de programmation ?
- des ressources machine (CPU, mémoire, ...)

Les outils du jour:

- Mesurer le temps d'exécution du programme implémentant l'algorithme (module time)
- Tracer une courbe (module matplotlib)
- Le module tqdm

[1]:
```
from time import time, sleep

debut = time()
# votre code ici
# sleep(2)
fin = time()
duree = fin - debut
print("Duree: ", duree)
```

```
Duree:  3.0040740966796875e-05
```

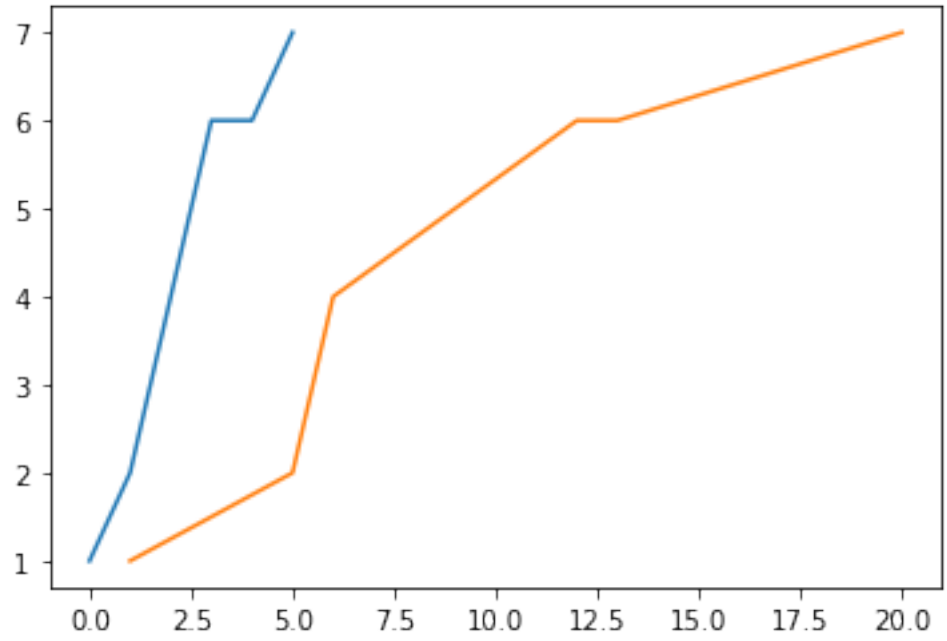
[2]:
```
def max(T):
    maxi = T[0]
    for Ti in T[1:]:
```

```
        if maxi < Ti:
            maxi = Ti
    return maxi
```

[3]:
```
from time import time, sleep
from random import randint

T, n = [], 10_000_000
for _ in range(n):
    T.append(randint(1, 100_000))

debut = time()
_ = max(T)
fin = time()
duree = fin - debut
print("Duree: ", duree)
```

```
Duree:  0.3929941654205322
```

[4]:
```
import matplotlib.pyplot as plt  # a installer

y = [1, 2, 4, 6, 6, 7]
x1 = list(range(len(y)))
x2 = [1, 5, 6, 12, 13, 20]

plt.plot(x1, y)
#plt.show()
plt.plot(x2, y)
plt.show()
```



[5]:
```
# tqdm a installer
#from tqdm import tqdm                  # programme python classique
from tqdm.notebook import tqdm          # dans un notebook
#from tqdm.gui import tqdm              # pour thonny
from time import sleep
from random import randint

for i in tqdm(range(5), desc='Boucle sur i'):
    for j in tqdm(range(30), desc=f'Boucle sur j', leave=False):
        duree_sommeil = randint(1, 2)
        sleep(duree_sommeil)
```

```
Boucle sur i:   0%|          | 0/5 [00:00<?, ?it/s]
Boucle sur j:   0%|          | 0/30 [00:00<?, ?it/s]
Boucle sur j:   0%|          | 0/30 [00:00<?, ?it/s]
Boucle sur j:   0%|          | 0/30 [00:00<?, ?it/s]
Boucle sur j:   0%|          | 0/30 [00:00<?, ?it/s]
Boucle sur j:   0%|          | 0/30 [00:00<?, ?it/s]
```

Cours 2

Le cours du jour traite des points suivants:

- Fonctionnement des listes en Python
- Pourquoi parle t'on de liste au lieu de tableau

- Comment fonctionnent append et insert ?
- Quelle est leur efficacité respective ?

On va parler de technique…

2.8 Rappels liste

Une **liste** est une structure de données qui contient une séquence de valeurs.

Syntaxe:

`[<valeur_1>, <valeur_2>, ..., <valeur_n>]`

- Les valeurs ne sont pas nécessairement de même type.
- Une liste est une séquence

Extrait de la documentation

- **list.append(x)** - *Add an item to the end of the list. Equivalent to a[len(a):] = [x].*
- **list.extend(iterable)** - *Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.*
- **list.insert(i, x)** - *Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).*
- **list.remove(x)** - *Remove the first item from the list whose value is equal to x. It raises a ValueError if there is no such item.*
- **list.pop([i])** - *Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.*

2.9 Généralités sur la mémoire

- La mémoire peut être vue comme un long ruban avec des zones protégées
- Chaque case mémoire (un octet) dispose d'une adresse propre.
- Le stockage d'une valeur peut prendre plusieurs octets (Ex.: mot de 8 octets)
- Certaines valeurs peuvent être de type "adresse" (8 octets ?). On les représente symboliquement par une flèche.

2.9.1 Tableau en mémoire (vers. classique)

- Usuellement un ensemble de cases **contigues** en mémoire de **même taille**.
- On fait de l'arithmétique avec la taille d'un objet pour trouver la position d'un élément:

$$@T[i] = @T[0] + i * taille(element)$$

- Avantage: Accès direct (rapide) à un élément.
- Inconvénient: Pas idéal pour des mises à jour:
  - Ajout: décalage si pas à la fin
  - Insertion/ suppression: décalage/suppression.

2.9.2 Liste simplement (ou doublement) chainée (vers. classique)

- Usuellement un ensemble de cases **non contigues** en mémoire.
- Chaque élément connait son successeur (et prédecesseur si doublement chainée).
- Dispersion des éléments en mémoire.
- Une valeur spéciale indique la fin (et le début) de liste.

- Avantage: idéal pour des mises à jour (ajout, insertion, suppression)
- Inconvénient: Accès séquentiel à un élément (très lent si longue liste)

Tableau ou Liste:

- On doit faire un compromis entre efficacité de l'accès et des mises à jour.
- Dépend des langages de programmation (Python vs C)

2.10 Listes (= Tableaux) en Python

2.10.1 Tests préliminaires

[6]:
```
from tqdm.notebook import tqdm_notebook
import random
from time import time
n = int(10**5.5)
```

Sondage:

Après n appels à *append* ou *insert*, que prévoyez vous ?

- Append est plus rapide
- Insert est plus rapide
- Tous les deux prennent le même temps

[7]:
```
L = []
tps_append = []
for _ in tqdm_notebook(range(n)):
    val = random.randint(0, 100)          # nombre aléatoire
    debut = time()
    L.append(val)                         # append
    fin = time()
    tps_append.append(fin - debut)
```

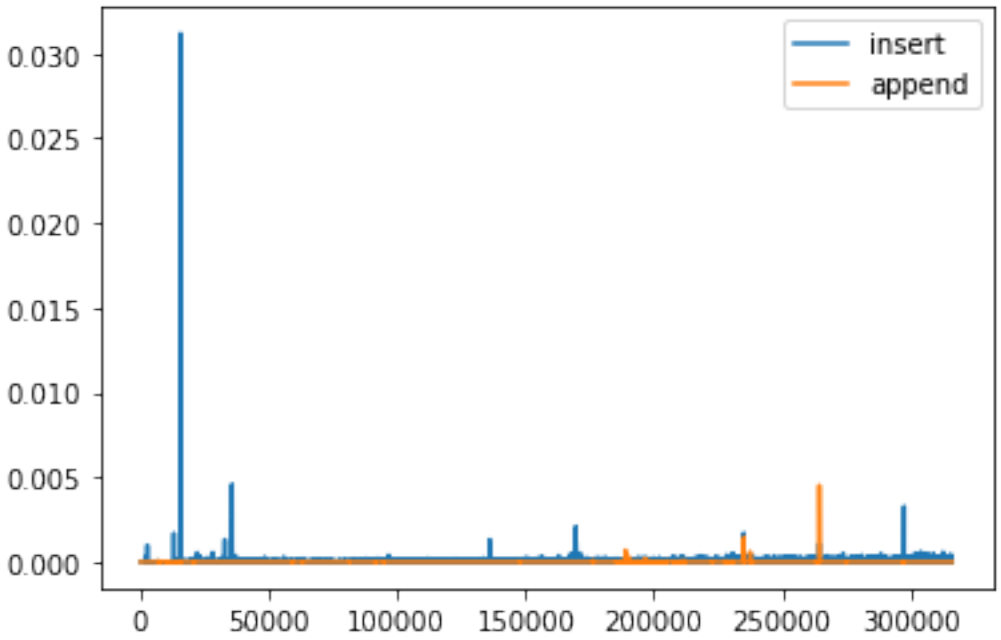
```
  0%|          | 0/316227 [00:00<?, ?it/s]
```

[8]:
```
L = []
tps_insert = []
for _ in tqdm_notebook(range(n)):
    pos = random.randint(0, len(L))       # position aléatoire
    val = random.randint(0, 100)          # nombre aléatoire
    debut = time()
    L.insert(pos, val)                   # insert
    fin = time()
```

```
    tps_insert.append(fin - debut)
```

```
  0%|          | 0/316227 [00:00<?, ?it/s]
```

[9]:
```
import matplotlib.pyplot as plt
x = list(range(n))

plt.plot(x, tps_insert, label = "insert")
plt.plot(x, tps_append, label = "append")
plt.legend()
plt.show()
```



Conclusions partielles:

- **append** bien plus rapide qu'**insert**
- **append**: le temps semble indépendent de la taille de la liste
- **insert**: le temps croît la taille de la liste

2.10.2 Liste et mémoire en Python

Une liste étant un objet, même une liste vide doit occupé de la place:

[10]:
```
from sys import getsizeof as sof    ## Nombre d'octets en mémoire
print(f"Entier: {sof(0)}")
print(f"Liste vide: {sof([])}")
```

```
Entier: 24
Liste vide: 56
```

Ne pas oublier que ce sont des objets avec des méthodes ...

[11]:
```
#dir([])
```

[12]:
```
# 8 octets par valeur de plus
sof([]), sof([1]), sof([1, 2]), sof([1, 2, 3])
```

[12]:
```
(56, 64, 72, 80)
```

[13]:
```
# Et ce quel soit le type et ses valeurs
sof([]), sof(["lepetitchat"]), sof(["le", "petit"]), sof(["le", "petit",␣
↪"chat"])
```

[13]:
```
(56, 64, 72, 80)
```

[14]:
```
# Même en mélangeant les types
sof([]), sof(["lepetitchat"]), sof(["le", ["petit"]]), sof(["le", ["petit"],␣
↪["chat", False]])
```

[14]:
```
(56, 64, 72, 80)
```

Python ne stocke donc pas directement les valeurs dans la liste mais des **références** vers les données:

- Liste = Tableau d'adresses vers les élements de la liste
- Le type des éléments de la liste n'est pas connue de la liste
- Cela optimise la place mémoire (intéressant pour la copie, pour **insert**)

Que faire si la taille initialement prévue est trop petite (insert ou append) ?

- Le système cherche un nouvel emplacement plus grand (peu couteux)
- Il recopie l'ancien contenu. De plus en plus couteux au fur et à mesure que le tableau croît.

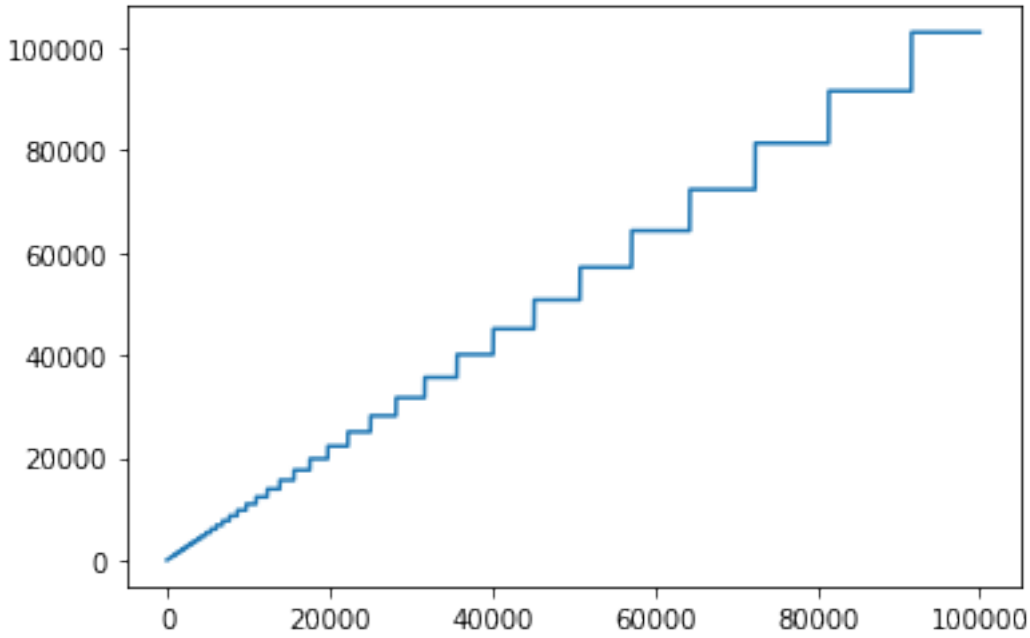
La croissance de la taille d'une liste est gérée par la règle:

```
new_allocated = newsize + newsize // 8 + 3 si newsize < 9 (+6 sinon)
new_allocated ~ 1.125 * newsize
```

Cela donne 0, 4=1+3, 8=5+3, 16=9+1+6, 25=17+2+9, ...

[15]:
```
import matplotlib.pyplot as plt
allocated = [0]
for size in range(1, 10**5):
    if size <= allocated[-1]:
        allocated.append(allocated[-1])
    else:
        newsize  = size + size // 8
        if size < 9:
            allocated.append(newsize+3)
```

```
        else:
            allocated.append(newsize+6)
plt.figure()
plt.plot(allocated)
plt.show()
```



Aparté: Principe identique pour les autres types

Octets	type	scaling notes
28	int	+4 bytes about every 30 powers of 2
49	str	+1-4 per additional character (depending on max width)
40	tuple	+8 per additional item
64	list	+8 per additional item
240	dict	6th increases to 368; 22nd, 1184; 43rd, 2280; 86th, 4704; 171st, 9320

[16]:
```
print(sof(0), sof(1), sof(2**30), sof(2**60), sof(2**90))
print(sof(""), sof("a"), sof("aa"), sof("aaa"), sof("aaaa"))
print(sof(tuple()), sof((1,)), sof((1,2)), sof((1,2,3)), sof((1,2,3,4)))
```

```
24 28 32 36 40
49 50 51 52 53
40 48 56 64 72
```

Que faire si la taille initialement prévue est trop grand ?

La décroissance de la taille d'une liste est gérée par la règle:

```
si new_size < allocated //2 alors on réalloue
sinon on garde l'allocation inchangée.
```

2.10.3 Ajout en tête

Quel est la meilleure solution pour le “prepend” (ajout en tête) rapide de plusieurs élements ?

- Insertion systématique en position 0. Non !
- Append pour chaque élement puis reverse à la fin.

2.10.4 Conclusion (le transparent le plus important du cours 2)

Quelles sont les méthodes lentes et rapides parmi:

L[index], insert(index, objet),

pop(), pop(index),

remove(valeur), L.append(valeur),

index(valeur)

Rapide: * L[index]: simple calcul arithmétique * L.append: ajout en fin + ev. réallocation (rare) * pop(): supprime en fin + ev. réallocation (rare)

Lent: * insert(index, objet) : décalage * index(valeur): parcours jusqu'a trouve * remove(valeur): parcours jusqu'a trouve + décalage (parcours entier) * pop(index) ou del L[index]: décalage

2.11 Compléments sur les listes

2.11.1 Listes en compréhension

```
L = [expression(x) for x in iterable if condition(x)]
```

où *iterable* est une séquence (liste, range, chaine, …)

- La condition est optionnelle.
- Le résultat est une liste.
- En général sur une seule ligne.

Quelques usages:

- Construire une liste à partir de *range*.
- Créer une liste à partir d'une autre (filtrage)
- Se substituer à un *for* simple.
- Une liste de listes.

2.11.2 Parcours simultané de plusieurs listes avec *zip*

```
zip(*iterables)
```

où *iterables désigne 0, 1, 2, … objets iterables (liste, chaine, dict, …)

- le type retourné est *zip* qui est iterable ;)

[17]:
```
type(zip())
```

[17]:
```
zip
```

[18]:
```
L1 = [2, 3, 5]
L2 = ['m', 'e', 't']

for (e1, e2) in zip(L1, L2):
    print(e1, '---', e2)
```

```
2 --- m
3 --- e
5 --- t
```

[19]:
```
for i, (e1, e2) in enumerate(zip(L1, L2)):
    print(i,':', e1, '---', e2)
```

```
0 : 2 --- m
1 : 3 --- e
2 : 5 --- t
```

On s'arrête sur la plus courte séquence:

[20]:
```
L1 = [2, 3, 5, 6, 1, 4]
L2 = ['m', 'e', 't']

for e1, e2 in zip(L1, L2):
    print(e1, '---', e2)
```

```
2 --- m
3 --- e
5 --- t
```

Autre approche possible (sans zip):

[21]:
```
L1 = [2, 3, 5, 6, 1, 4]
L2 = ['m', 'e', 't']

for i in range(min(len(L1), len(L2))):
    print(L1[i], '---', L2[i])
```

```
2 --- m
3 --- e
5 --- t
```

Cela devient difficile de faire mieux que:

[22]:
```
L1 = [2, 3, 5, 6, 1, 4]
L2 = ['m', 'e', 't', 'i', 'e', 'r']
L3 = [2**i for i in range(10)]
```

```
L4 = [True, True, False]*4

for e1, e2, e3, e4 in zip(L1, L2, L3, L4):
    print(e1, '---', e2, '---', e3, '---', e4)
```

```
2 --- m --- 1 --- True
3 --- e --- 2 --- True
5 --- t --- 4 --- False
6 --- i --- 8 --- True
1 --- e --- 16 --- True
4 --- r --- 32 --- False
```