Listes

Skander Zannad et Judicaël Courant

Lycée La Martinière-Monplaisir

2013-09-28

1 Tableaux

1.1 En programmation

Tableau : *structure de données*, stockant des informations, généralement indexées par des entiers en accès direct en lecture et écriture.

En général, un tableau est une suite de n emplacements mémoire consécutifs tous de même largeur, numérotées de 0 à $n-1$. Pour accéder au contenu de l'emplacement numéro k, la machine a seulement besoin de connaître l'adresse de la première case mémoire et de la largeur de chaque case (dessin).

1.2 En Python

Les tableaux sont appelés des *listes*. Pour créer un tableau on peut :

1. Le donner en extension :

```
>>> t = [23, 41, 101]
>>> t
[23, 41, 101]
>>> t[0]
23
>>> t[1]
41
>>> t[2]
101
```

Attention à utiliser un indice existant :

```
>>> t[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Les indices appartiennent à $[\![0, \texttt{len(t)}[\![$.

Mais on peut aussi compter les éléments à partir de la fin :

```
>>> t[-1] # dernier élément
101
>>> t[-2] # avant-dernier
41
>>> t[-3]
23
>>> t[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

2. Le construire à partir de tableaux existants par concaténation :

```
>>> t = [13, 23, 43]
>>> u = [23, 41, 101]
>>> t + u
[13, 23, 43, 23, 41, 101]
>>> t * 2
[13, 23, 43, 13, 23, 43]
>>> 2 * t
[13, 23, 43, 13, 23, 43]
```

3. Le construire à partir de tableaux existants par tranchage (*slicing*) :

```
>>> t = [13, 23, 43, 53, 41, 101]
>>> t[2:5]
[43, 53, 41]
>>> t[3:]
[53, 41, 101]
>>> t[:3]
[13, 23, 43]
>>> t[1:5:2]
[23, 53]
```

2 Notion de complexité

2.1 Cadre : recherche du plus petit élément

Écrire une fonction `maxi` donnant le plus grand élément d'un tableau passé en argument.

Comment écrire `maxi` ?

Première version :

```
def est_majore_par(t, M):
    """Retourne un booléen indiquant si les éléments
    de la liste t sont majorés par M."""
    for x in t:
        if x > M:
            return False
    return True
```

Il ne reste plus qu'à trouver un élément du tableau le majorant :

```
def maxi1(t):
    """Retourne le plus grand élément de la liste t.
    t doit être non vide"""
    assert t != []
    for M in t:
        # invariant : aucun des éléments précédent
        # n'était le maximum
        if est_majore_par(t, M):
            return M
    # si t non vide, on n'arrive jamais ici :
    assert False
```

2.2 Complexité en temps

La *complexité en temps* d'un algorithme est le temps qu'il faut pour exécuter cet algorithme en fonction d'un *paramètre*, en général la taille de l'entrée.

Pour `maxi1`, on s'intéressera au temps mis en fonction du nombre d'éléments de `t`.

2.3 Étude expérimentale

On essaie la fonction sur cent listes de longueurs 1 à n :

1. Avec des listes de la forme `range(1,n)`
2. Avec des listes de la forme `range(n, 0, -1)`
3. Avec des listes contenant des valeurs aléatoires prises dans $[\![0, 10^9[\![$.

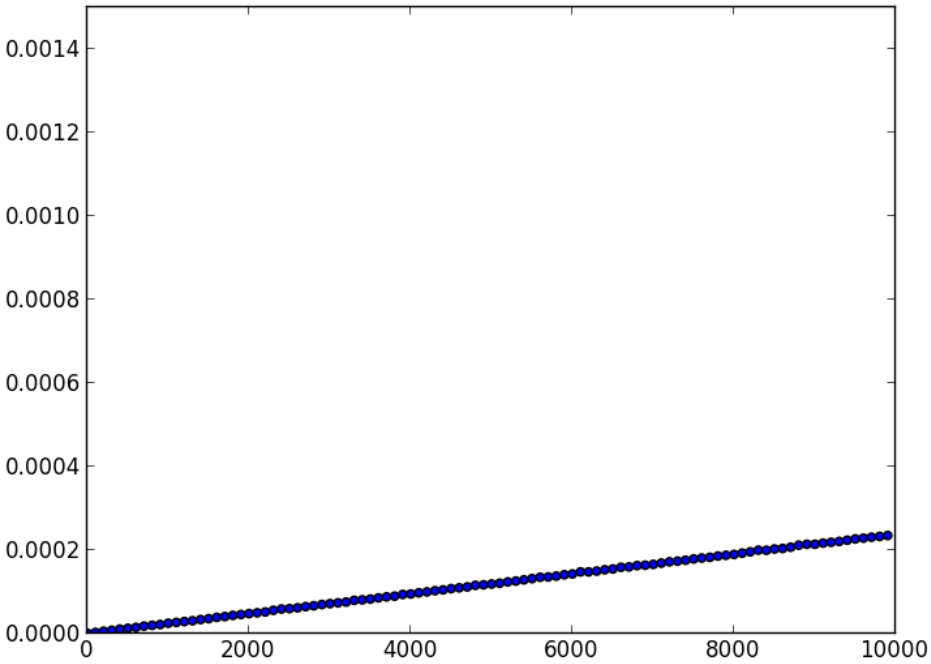

FIGURE 1: $n = 10\,000$, `range(n, 0, -1)`

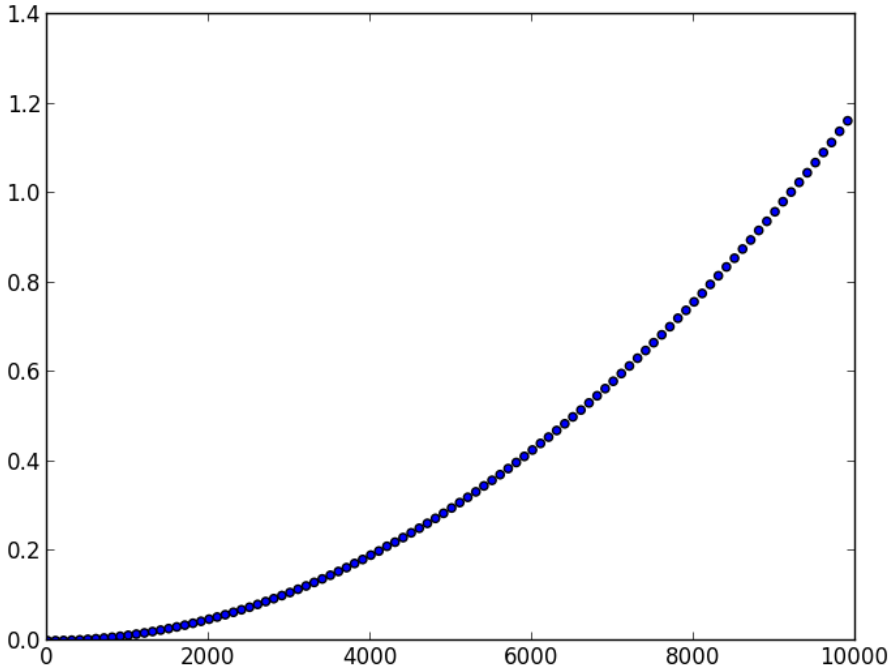

FIGURE 2: $n = 10\,000$, `range(n)`

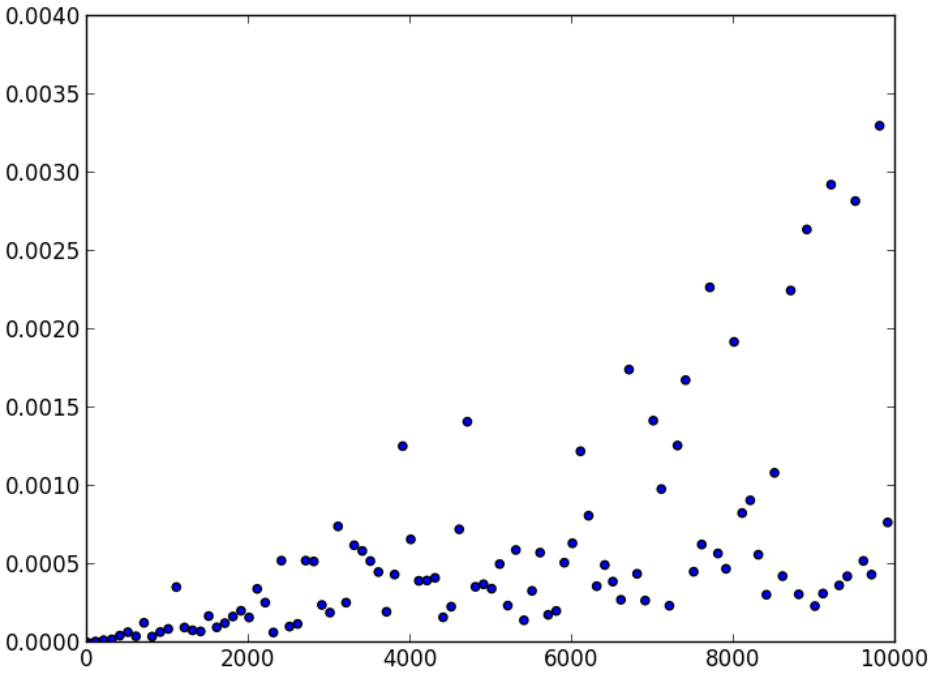

FIGURE 3: $n = 10\,000$, randlist

Conclusion ?

Et si on essaie avec $n = 1\,000\,000$?

Et pour `range(n)` pour $n = 1\,000\,000$:

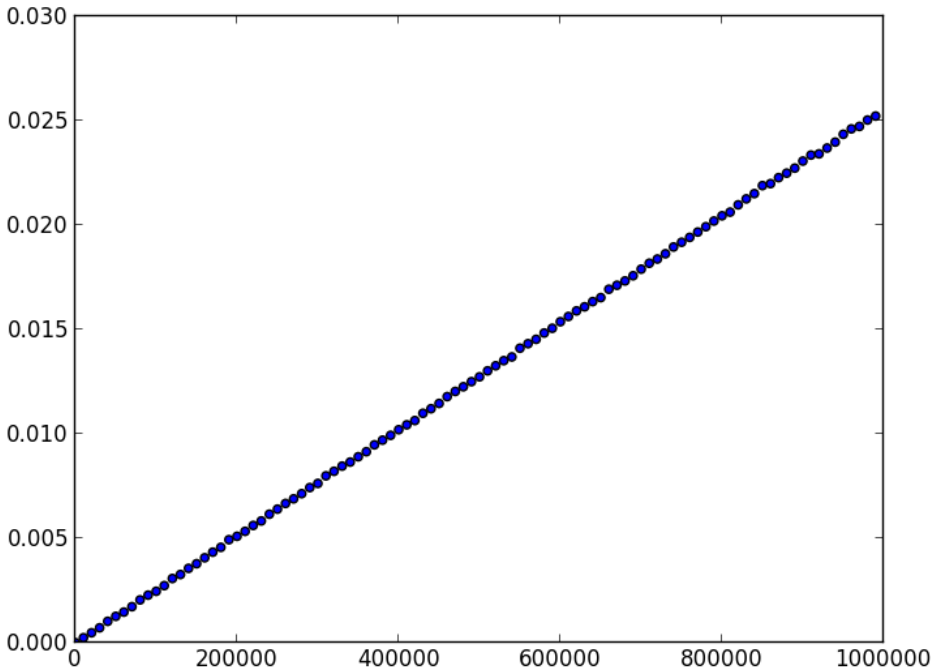

FIGURE 4: $n = 1\,000\,000$, `range(n, 0, -1)`

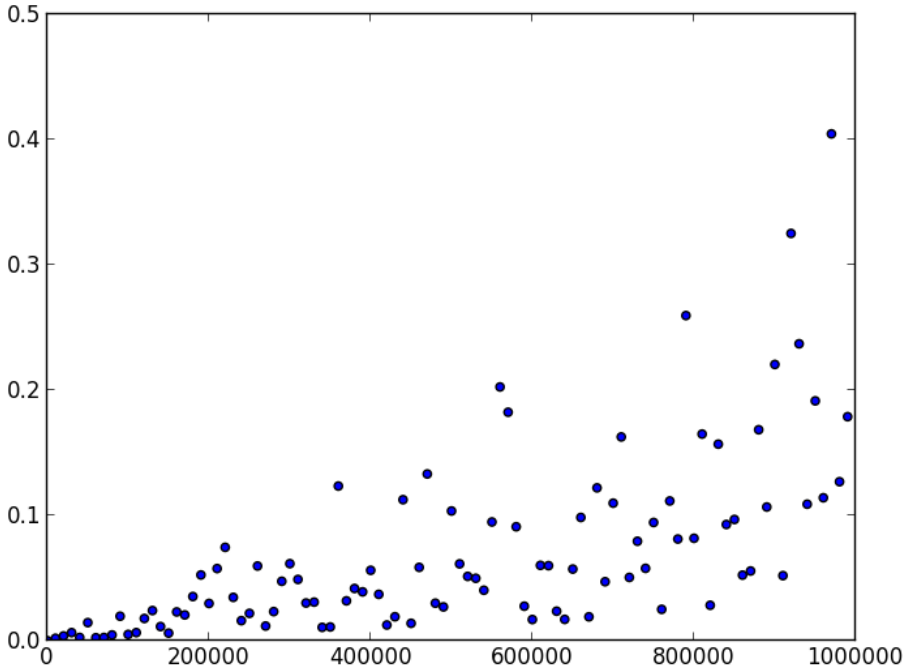

FIGURE 5: $n = 1\,000\,000$, randlist

De l'ordre de $10^4 s$, soit de l'ordre de $3h$.

Et pour $n = 10^9$?

Dans les différents cas, de l'ordre de

1. 22s
2. 270s
3. 10^{10}s soit 7 à 8 milliers d'années...

2.4 Étude théorique

Idées importantes :

1. Il est difficile d'estimer précisément le temps requis.
2. On peut essayer d'estimer le comportement dans le meilleur des cas, dans le cas le pire et en moyenne.
3. Le comportement en moyenne est en général délicat à calculer (amis des probas...)
4. On se contentera dans ce cours d'essayer d'estimer la complexité dans le cas le meilleur et le cas le pire.

Remarque : l'étude théorique ne dispense pas des tests pratiques.

2.5 Un majorant du temps utilisé

`est_majore_par(t, M)` (en notant n la longueur de t) :

- au plus n tours de boucle ;
- une comparaison dans chaque boucle ;
- temps de chaque comparaison supposé indépendant de M et x.

Temps mis : au plus $an + b$ où a et b sont des constantes indépendantes de n.

`maxi1(t)` (n longueur de t) :

- au plus n tours de boucle ;
- un appel à `est_majore_par` dans chaque boucle ;
- temps de chaque tour de boucle : majoré par $a'n + b'$ où a' et b' constantes indépendantes de n.

Temps mis : au plus $a'n^2 + b'n + c'$, où a' et b' constantes indépendantes de n.

Donc le temps mis peut être majoré (à partir d'un certain rang) par un Cn^2 où C est une constante.

On dit que ce temps est un $O(n^2)$ (« grand O de n^2 »).

2.6 Un minorant pour les tableaux strictement croissants

Regardons le cas où `t` est un tableau de taille n trié par ordre strictement croissant.

- exactement `len(t)` tours de boucle dans `maxi1(t)` ;
- `est_majore(t, x)` demande un temps $ak + b$ si x est l'élément d'indice k de t (avec $k < n - 1$) ;
- `est_majore(t,x)` demande un temps $cn + d$ si x est le dernier élément.

Or

$$\sum_{k=0}^{n-2}(ak+b) = a\frac{(n-2)(n-1)}{2} + b(n-1)$$

Donc le temps mis est minoré par un $C'n^2$ (à partir d'un certain rang), avec $C' > 0$.

On dit que c'est un $\Omega(n^2)$.

2.7 Cas le pire : analyse asymptotique

- Le temps mis est compris entre $C'n^2$ et Cn^2 avec $C' > 0$. On dit que ce temps est un $\Theta(n^2)$.
- Les constantes varient suivant la machine, le langage utilisé, etc. donc sont difficiles à donner.
- L'essentiel est de savoir que «quand n tend vers l'infini ça se comporte en gros comme du n^2».

2.8 Cas le meilleur

- Si x est un maximum, `est_majore_par(t, x)` parcourt tout le tableau. Temps minoré par un $\Omega(n)$.
- Il y a toujours au moins un maximum, donc `maxi1(t)` fait au moins un appel à `est_majore_par` avec le maximum. Donc `maxi1(t)` a toujours un temps minoré par un $\Omega(n)$.
- Dans le cas d'une liste où le premier élément est maximal, `maxi1(t)` fait un unique appel à `est_majore_par` (de complexité $O(n)$). Temps de `maxi1(t)` majoré par un $O(n)$.

Conclusion : dans le cas le meilleur, le temps de calcul est un $\Omega(n)$.

3 Complexités usuelles

Grosso modo :

- une opération élémentaire : 10^{-9}s à 10^{-7}s
- une journée : $8,6 \times 10^4$s
- une semaine : 6×10^5s
- un mois : $2,6 \times 10^6$s
- un an : $3,16 \times 10^7$s

Moyen mnémotechnique : πs $=$ un nanosiècle.

n	temps pour une complexité					
	$\log n$	n	$n \log n$	n^2	n^3	2^n
1		10^{-9}s		10^{-9}s	10^{-9}s	2.10^{-9}s
10	10^{-9}s	10^{-8}s	10^{-8}s	10^{-7}s	10^{-6}s	10^{-6}s
100	2×10^{-9}s	10^{-7}s	2×10^{-7}s	10^{-5}s	10^{-3}s	10^{30}s
1000	3×10^{9}s	10^{-6}s	3×10^{-6}s	10^{-3}s	1s	
10^4	4×10^{9}s	10^{-5}s	4×10^{-5}s	10^{-1}s	10^3s	
10^5	5×10^{9}s	10^{-4}s	5×10^{-4}s	10s	10^6s	
10^6	6×10^{9}s	10^{-3}s	6×10^{-3}s	10^3s	10^9s	
10^9	9×10^{9}s	1s	9s	10^5s	10^{12}s	