Algorithmes de tri... et un peu de numpy/scipy

Vendredi 20 novembre 2015

Buts du TP

- Se (re)mettre en tête un peu de numpy/scipy.
- Coder effectivement les algorithmes de tris vus en cours.
- Tester expérimentalement la complexité.

Dans tout ce TP, les tableaux sont implicitement constitués d'entiers ou de flottants.

Exercice 1. *Créer (au bon endroit) un dossier associé à ce TP.*
Lancer Spyder, sauvegarder immédiatement au bon endroit le fichier `.py` *; écrire une commande absurde, de type* `print(5*3)` *dans l'éditeur, sauvegarder et exécuter. Vérifier (F6) les options d'exécution. Je recommande d'imposer un nouvel interpréteur, et garder la main dessus (il y a donc deux cases à vérifier/cocher).*

Récupérer sur le réseau le fichier `cadeau_tris.py` *; Observer et comprendre le contenu.*

On conseille de tester les différents tris sur des tableaux de taille « parlante » : 100 puis 1000 puis 10000 (pour les tris en $n \ln n$) ou 2000 puis 4000 et observer les rapports des temps d'exécution. Lorsque n est doublé (resp. multiplié par 10), si le temps est multiplié par 4 (resp. 100), ça donne une petite idée du type de complexité...

Je rappelle qu'il est IMPORTANT de copier/coller les résultats de la fenêtre d'exécution VERS la fenêtre d'édition. Vous n'avez pas compris la phrase précédente ? Alors il est urgent de commencer à apprendre à travailler : je vous suggère de demander des précisions MAINTENANT, et pas vers le 15 juin. À bon entendeur...

1 Un peu de numpy/scipy (pas plus de 40 minutes)

Exercice 2. *Demander à Python ce que vaut* $\displaystyle\int_0^1 t^2 dt$.
On commencera par définir la fonction $t \mapsto t^2$ (dur : il faut commencer par choisir un nom...), puis on utilisera la fonction `quad` de la librairie `scipy.integrate`

La période d'un pendule libre lancé depuis un angle θ_0 vaut :

$$T(\theta_0) = \sqrt{\frac{\ell}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos\theta - \cos\theta_0}},$$

avec ℓ la longueur du pendule et g ce que vous imaginez.

Exercice 3. *Que vaut* $T(\pi/4)$ *? (on prendra* $\ell = 0.1$ *et* $g = 10$*)*
Représenter le graphe de T *sur l'intervalle* $[0.01, 0.95\pi]$ *:*

```
from matplotlib.pyplot import *
from numpy import linspace

les_t = linspace(0.01, 0.95*pi, 200)
les_y = [T(t) for t in les_t]

plot(les_t, les_y)
grid()

savefig('periode_pendule.pdf')
```

On s'intéresse maintenant au spectre des matrices stochastiques. On peut créer une telle matrice stochastique « aléatoire » en définissant d'abord une matrice à coefficients dans $[0, 1]$... puis en normalisant : on divise chaque ligne par la somme des coefficients de cette ligne !

```
from random import random

def aleatoire(n):
    A = array([[random() for _ in range(n)] for _ in range(n)])
    for i in range(n):
        somme = sum(A[i , :])
        for j in range(n):
            A[i,j] = A[i,j] / somme
    return A
```

Exercice 4.
- *Comprendre PUIS taper les lignes précédentes ; tester cette fonction.*
- *Sur différentes matrice stochastiques* $(5, 5)$*, calculer le spectre avec* `eigvals`*, déterminer le module des valeurs propres (par exemple :* `modules = [abs(lamb) for lamb in spectre]`*) et vérifier ce que dit la théorie !*

2 Tris « sélection » (pas plus de 20 minutes)

On commence par le « tri-bulles ». Il s'agit d'un tri « en place » : le programme reçoit un tableau sur lequel il agit, mais ne retourne rien.

Entrées : T
pour i *de* $|T| - 1$ *à* 1 **faire**
  pour j *de* 0 *à* $i - 1$ **faire**
    si $T[j] > T[j + 1]$ **alors**
      $T[j] \leftrightarrow T[j + 1]$

Exercice 5. Tri-bulles
Programmer effectivement le tri-bulles ; le tester et le chronométrer. Noter les résultats puis commenter éventuellement la partie du script qui lance les tests ou contient les résultats chronométrés.

Avant de coder le tri par sélection classique, quelques fonctions préliminaires (seule la dernière sera effectivement utilisée dans la suite). Il s'agit essentiellement de s'échauffer le cerveau, et de se poser les bonnes questions :
- Pour tel type de problème, de quelles variables vais-je avoir besoin ?
- Que vont-elles représenter ?
- Comment vais-je les initialiser ?
- Comment vont-elles évoluer en cours de route ?

Exercice 6. *Écrire des fonctions calculant et renvoyant respectivement :*
- *la valeur maximale d'un tableau non vide ;*
- *la (une) position du maximum ;*
- *la (une) position du maximum parmi les k premières cases d'un tableau (la fonction reçoit le tableau et k).*

Tester les programmes (sur des valeurs mettant potentiellement votre programme en difficulté, sinon ça ne sert à rien !)

Voici enfin l'algorithme du tri par sélection, qui est à nouveau un tri en place :

Entrées : T
pour j *de* $|T|$ *à* 2 **faire**
  Trouver la position p du maximum des j premiers éléments du tableau
  $T[p] \leftrightarrow T[j - 1]$

Exercice 7. *Coder effectivement le tri par sélection. Le tester et le chronométrer.*

Exercice 8. *Au chronomètre, comparer la complexité dans les cas moyens et celle dans les cas déjà triés.*

3 Tris « insertion » (pas plus de 20 minutes)

Dans les tris par insertion, on place chaque nouvel arrivant dans le tableau à sa position, dans une zone déjà triée : par exemple dans l'exemple suivant, on doit insérer 6 dans un sous-tableau déjà trié :

0	5	**9**	**10**	**13**	**6**	*	*

$\longrightarrow$

0	5	**6**	**9**	**10**	**13**	*	*

Pour réaliser l'opération précédente (décalage de $T[5]$ vers la position 2) on peut écrire

```
for j in range(5, 2, -1):
    T[j-1], T[j] = T[j], T[j-1]
```

Mais on peut aussi brutaliser, à l'aide du slicing offert par Python :

```
T[2: 6] = [ T[5] ] + T[2: 5]
```

Pour insérer $T[i]$ dans $T[:i]$, il faut commencer par *trouver la position* de ce nouvel élément $T[i]$; par exemple comme ceci :

Entrées : T, i
$pos \leftarrow i$ # la position d'affectation ; au départ : à droite !
tant que $pos > 0$ *et* $T[pos-1] > T[i]$ **faire**
  $pos \leftarrow pos - 1$
Résultat : pos

Exercice 9. *Écrire une fonction prenant en entrée un tableau T et un indice $i \in [\![1, |T|-1]\!]$ et retournant la position où on doit insérer $T[i]$ dans $T[:i]$ supposé déjà trié.*

On peut maintenant écrire le tri par insertion :

Entrées : T
pour i *de* 1 *à* $|T|-1$ **faire**
  $pos \leftarrow position(T, i)$
  décaler $T[i]$ en position pos

Exercice 10. *Programmer effectivement le tri par insertion. Le tester et le chronométrer, en particulier dans les cas déjà triés.*

On rappelle la variante suivante, qui consiste à trouver la position d'insertion par une recherche dichotomique de la position :

Entrées : T, i
si $T[i] < T[0]$ **alors**
  Résultat : 0
si $T[i] > T[i-1]$ **alors**
  Résultat : i
$g, d \leftarrow 0, i-1$ # les bords gauche et droit de l'intervalle de recherche
tant que $g < d-1$ # *On s'assure :* $d \geqslant g+2$, *et* $T[g] \leqslant T[i] \leqslant T[d]$ **faire**
  $m \leftarrow (g+d)//2$ # l'indice du milieu ; $g < m < d$
  si $T[m] \leqslant T[i]$ **alors**
    $g \leftarrow m$ # continuer à droite
  sinon
    $d \leftarrow m$ # continuer à gauche
Ici, $d = g+1$, et $t[g] \leqslant t[i] \leqslant t[d]$
Résultat : d

Exercice 11. *Programmer cette fonction d'insertion ; tester et chronométrer le nouveau programme de tri par insertion ainsi obtenu.*

4 Tri « fusion »

On rappelle le principe du tri *fusion* ou *dichotomique* : si un tableau a une taille $\geqslant 2$, on le casse en deux, on trie récursivement les deux morceaux T_1 et T_2, puis on les fusionne via l'algorithme suivant, où on adjoint (via la méthode `append`) à un tableau initialement vide successivement les éléments de T_1 et T_2 :

Entrées : T_1, T_2
$Res \leftarrow []$ # le tableau qui sera rendu
$i_1, i_2 \leftarrow 0, 0$ # les indices qu'on regarde dans les tableaux T_1 et T_2
tant que $i_1 < |T_1|$ *ou* $i_2 < |T_2|$ **faire**
  si $i_2 = |T_2|$ *ou* $(i_1 < |T_1|$ *et* $T_1[i_1] < T_2[i_2])$ **alors**
    Adjoindre $T_1[i_1]$ à Res
    $i_1 \leftarrow i_1 + 1$
  sinon
    Adjoindre $T_2[i_2]$ à Res
    $i_2 \leftarrow i_2 + 1$
Résultat : Res

Exercice 12. *Écrire une fonction réalisant la fusion de deux tableaux supposés triés.*

C'est presque fini !

Exercice 13. *Écrire une fonction réalisant le tri-fusion d'un tableau. Cette fonction doit renvoyer un nouveau tableau (et non pas trier en place comme dans les deux premières parties).*

Exercice 14. *Tester et chronométrer le tri fusion, y compris dans les cas déjà triés.*

5 Tri rapide

Le principe du tri rapide consiste, pour trier une zone $[\![d, f]\!]$ d'un tableau, si $f > d$, à effectuer une première phase de préparation de la zone, à l'issue de laquelle les éléments de cette zone ont été permutés pour avoir le schéma :

$*$	$T[d] = p$	...	$T[f]$	$*$	$\longrightarrow$	$*$	$\leqslant p$	p	$> p$	$*$

En plus de la préparation de cette zone, l'algorithme doit retourner la nouvelle position du pivot (ce qui sera crucial pour lancer les deux appels récursifs ultérieurs pour trier la zone).

Un algorithme permettant de préparer la zone $[\![d, f]\!]$ consiste à manipuler deux indices i_1 et i_2 délimitant les fins respectives de zones d'éléments respectivement *majorés par* et *strictement minorés par* le pivot $p = T[d]$:

$*$	p	$\leqslant p$	$T[i_1]$	$> p$	$T[i_2]$	$*$	$*$

L'invariant préservé est :

$$(d < i \leqslant i_1 \implies T[i] \leqslant p) \quad \text{et} \quad (i_1 < i \leqslant i_2 \implies T[i] > p)$$

Entrées : T, d, f
$i_1, i_2 \leftarrow d, d$ # Si, vraiment !
tant que $i_2 < f$ **faire**
  si $T[i_2 + 1] \leqslant p$ **alors**
    $T[i_1 + 1] \leftrightarrow T[i_2 + 1]$
    $i_1 \leftarrow i_1 + 1$
  $i_2 \leftarrow i_2 + 1$
$T[d] \leftrightarrow T[i_1]$
Résultat : i_1

On note que cet algorithme est à « effets de bord » (il modifie le tableau qui a été donné) et retourne également un résultat (l'indice du pivot à la fin).

Exercice 15. *Écrire une fonction réalisant ce travail de préparation. La tester.*

L'élément principal du tri rapide est écrit ; terminons le travail !

Exercice 16. *Écrire une fonction réalisant en place le tri rapide d'un tableau ; tester et chronométrer.*
On observera en particulier le comportement sur les tableaux déjà triés.

S'il vous reste du temps : mettez au point et programmez l'algorithme de préparation de tableau (dans le tri rapide) consistant à maintenir l'invariant décrit ainsi :

$*$	p	$\leqslant p$	$T[i_1]$	$*$	$T[i_2]$	$> p$	$*$

On réfléchira à l'initialisation de i_1 et i_2, au principe leur permettant de se rejoindre... puis on implémentera le tout.

Au chronomètre, j'obtiens des temps légèrement meilleurs (de l'ordre de 15%) qu'avec la première méthode.