TP5 : tris évolués

AP3 Algorithmique et programmation — L2 mathématiques

20 octobre 2016

On rappelle une façon simple de générer des listes aléatoires :

```
import random # library with function about randomization

# random contains a function randrange(n,m) returning a random
    integer between n and m-1
li = [random.randrange(1,100) for i in range(20)]
```

Exercice 1 (Tri par insertion (reprise du TD4))**.** Soit `li` une liste de longueur n. Le tri par insertion est une variante des tris précédents. Il consiste, à chaque phase $i \leq n-1$, à insérer la valeur d'indice i à sa place parmi les valeurs d'indices 0 à $i-1$ qui sont supposées être déjà triées en ordre croissant mais ne sont pas nécessairement les $i-1$ plus petites valeurs de la liste.

Détaillons un peu. Une liste à un seul élément est triée par définition. Pour $i = 2$, on cherchera à positionner `li[1]` correctement vis à vis de `li[0]` de façon à obtenir une liste triée constituée de ces deux éléments. Plus généralement, pour $i \leq n-1$, on supposera que les $i-1$ premiers éléments de `li` ont été ré-ordonnés de façon à ce que l'on ait `li[0]` $\leq$ `li[1]` $\leq \ldots \leq$ `li[i-1]` ; et on insère `li[i]` à sa place dans cette liste. Lorsqu'on arrive à $i = n$, on a `li[0]` $\leq$ `li[1]` $\leq \ldots \leq$ `li[n-1]`, c'est-à-dire que la liste est triée.

Toute la difficulté est alors dans l'*insertion* de `li[i]` dans la sous-liste triée `li[:i]` : on compare `li[i]` et `li[i-1]` et on les échange si besoin ; puis on recommence avec `li[i-1]` et `li[i-2]` etc. jusqu'à ce qu'un échange ne soit plus nécessaire. L'élément initialement en position i est alors inséré à la bonne position, faisant de la sous-liste `li[:i+1]` une liste triée.

(1) Écrire une fonction `insert_sort(li)` prenant en paramètres une liste `li` implémentant le tri par insertion rappelé ci-dessus.

(2) On peut améliorer légèrement la fonction ci-dessus, en insérant l'élément `li[i]` dans `li[:i]` par recherche dichotomique. On rappelle que la recherche dichotomique **dans une liste triée** se fait comme suit : on a un élément à insérer à sa place dans la liste triée ; on le compare à l'élément du milieu(*-ish*) de la liste ; s'il est plus petit, on répète récursivement l'opération sur la sous-liste de gauche, sinon sur celle de droite ; les cas d'arrêt sont le cas d'une liste vide et le cas où l'on tombe tout pile sur la valeur qu'on essaye d'insérer.

(3) Cette optimisation peut être utile si une comparaison est coûteuse par rapport à l'échange de deux éléments. Qu'entendons-nous par cette phrase ?

Exercice 2 (Tri fusion)**.**

(1) Écrire une fonction `merge(li1,li2)` prenant en paramètres deux listes triées `li1` et `li2` et renvoyant la fusion de ces deux listes. On rappelle que la fusion se fait comme suit : on compare les premiers éléments de chaque liste et on sélectionne le plus petit des deux et on l'enlève à sa liste d'origine ; on recommence sur les deux listes restantes (l'une étant donc maintenant amputée de son premier élément) etc. jusqu'à ce qu'une des deux listes soit vide ; on concatène alors le résultat avec la liste restante.

(2) Déduire de la question précédente une fonction `merge_sort(li)` prenant en paramètres une liste `li` et la renvoyant triée par la méthode du tri fusion. La méthode, vue en cours, se résume à ceci : on coupe la liste en deux parties égales, qu'on trie séparément (et récursivement) par la méthode de tri fusion ; puis on fusionne les deux listes ainsi obtenu.

Exercice 3 (Tri par tas)**.** On rappelle les notions suivantes. Un tableau d'entiers A indexé de 1 à n est organisé *en tas à la position* i ($1 \leq i \leq n$) si et seulement si l'une des trois conditions suivantes est satisfaite,

- $2i > n$
- $2i = n$ et $A[i] \geq A[2i]$
- $2i < n$ et $A[i] \geq \max(A[2i], A[2i+1])$ et A est organisé en tas aux positions $2i$ et $2i+1$.

Un tableau est organisé en tas s'il est organisé en tas à toutes ses positions.

(1) Écrire une fonction `heap_at(li,p,end)` qui prend en paramètres une liste `li` telle que `li[:end]` est en tas aux indices `2p` et `2p+1` (si ceux ci sont tous les deux strictement inférieurs à `end`), un indice $p \leq \lfloor n/2 \rfloor$ et un indice de fin `end` et qui renvoie la liste `li` modifiée de façon que `li[:end]` soit en tas à l'indice `p`. (C'est-à-dire que l'on fait comme si les élements de `li` à partir de l'indice `end` n'avaient pas d'importance ; ce qui permettra par la suite d'appliquer cette fonction à des sous-listes de `li`.)

(2) En déduire une fonction `heap(li)` prenant en paramètres une liste `li` et modifiant celle-ci de façon à ce qu'elle soit organisée en tas (i.e. elle est *en tas* à tous les indices).

(3) Écrire une fonction `heap_sort(li)` triant la liste `li` selon la méthode du tri par tas.

Exercice 4. Comparer expérimentalement les performances des différents tris implémentés. Pour ce faire, générer aléatoirement des listes d'entiers de tailles importantes (on pourra commencer avec des listes de taille 100) et comparez le temps d'exécution des différentes fonctions de tris (on pourra utiliser le package `timieit` de `python` par exemple).

Attention : étant donnée les limitations du langage `python` concernant la récursivité, on évitera de tester sur des listes de taille 1000 et plus, au risque d'avoir des erreurs un peu obscures du type : `Maximum recursion depth reached`.

À rendre pour le 3 novembre 2016

Exercice 5.

(i) Écrire une fonction, la plus naïve possible, renvoyant la somme maximale que l'on peut obtenir en additionnant des éléments consécutifs d'une liste `li` :

```
def max_sum (li): #...
```

Par exemple, la fonction doit renvoyer `11` sur la liste `[-1,1,8,-7,9,-2]` (réalisé par les éléments consécutifs $1, 8, -7, 9$).

(ii) Estimer la complexité de la fonction précédente (en fonction de la taille de la liste).

(iii) Le but de cette question est d'obtenir un algorithme plus efficace. Notons m_i la somme maximale d'éléments consécutifs pour la sous-liste `li[:i+1]` allant de l'indice 0 à l'indice i. Par exemple, m_0 est 0 (la somme vide) si `li[0]` est négatif et vaut `li[0]` sinon.

En exprimant m_{i+1} en fonction de m_i, écrire une fonction calculant la même chose qu'en question (a) mais plus efficacement :

```
def max_sum_better (li): #...
```

(iv) Quelle est la complexité de cette nouvelle fonction ?

Cet exercice est une initiation à une technique que vous allez beaucoup utiliser par la suite : la *programmation dynamique*.