TP5 bis : tri en tas, tri rapide

AP3 Algorithmique et programmation — L2 mathématiques

3 novembre 2016

Exercice 1 (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`, un indice `p` et un indice de fin `end` telle que `li[:end]` est en tas aux indices $2p$ et $2p+1$ (si ceux ci sont tous les deux strictement inférieurs à `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 : on met le tableau en tas et on sait alors que le maximum se trouve à la première position ; on le sélectionne et on recommence sur le tableau privé de cette première position etc. jusqu'à tomber sur un tableau à un unique élément.

(4) Estimer la complexité du tri par tas.

Exercice 2 (Tri rapide)**.** Tout comme le tri fusion, le tri rapide utilise une méthode *diviser-pour-régner*. Mais au contraire du tri fusion qui fait ses appels récursifs immédiatement pour fusionner ensuite deux listes triées, le tri rapide commencent par faire une répartition grossière de deux sous-listes puis effectue ses appels récursifs.

Plus précisément, on choisit une case de la liste p, appelé *pivot*. On met tous les éléments plus petits que p à sa gauche, et tous les éléments plus grand à sa droite. Puis on rappelle récursivement la fonction de tri rapide pour traiter les deux sous-tableaux ainsi créés.

(i) Écrire une fonction `quick_sort(li)` qui implémente le tri rapide. En particulier, on n'hésitera pas à créer deux nouvelles listes (celle des éléments plus petits et celle des éléments plus grands que le pivot) à chaque appel récursif.

(ii) Estimer la complexité en temps du tri rapide. Estimer également sa complexité **en espace** (i.e. une borne sur la place prise en mémoire lors de l'exécution de l'algorithme).

(iii) *(Bonus).* On peut améliorer la complexité en espace en effectuant le tri *en place* : c'est-à-dire qu'au lieu de créer deux listes à chaque appel récursif, on réorganise la liste elle-même de manière à avoir les éléments plus petits que le pivot à sa gauche et les plus grands à sa droite.

Commencer par écrire une fonction `distribute(li,p)` qui prend en paramètre une liste `li` et le pivot p, et qui modifie `li` par des échanges successifs, de telle façon qu'à la fin, les éléments plus petits que le pivot soit à sa gauche et les éléments plus grands à sa droite. On n'hésitera pas à parcourir la liste simultanément depuis le début et depuis la fin.

En déduire une fonction `quick_sort_inplace(li, start, end)` qui effectue un tri rapide en place de la sous-liste `li[start:end]`. En particulier, l'appel `quick_sort_inplace(li,0,len(li))` trie la liste `li` entière.

Exercice 3. 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`.