Université Paris 7 - Licence d'informatique - L2

Eléments d'Algorithmique

Examen du 19 Mai 2008 - durée : 3h00

Tous les algorithmes proposés devront être commentés en expliquant leur principe de fonctionnement. Si nécessaire, expliquer également avec un petit commentaire en francais, chaque instruction individuelle qui serait moins évidente à comprendre pour le correcteur. Les algorithmes vus en cours ne doivent pas être détaillés.

Notes de cours manuscrites autorisées. Livres non autorisés.

Il va sans dire que la rigueur des raisonnements, la clarté des explications, mais aussi la qualité de la présentation influera sensiblement sur la note.

Le barème n'est donné qu'à titre indicatif et pourra donc être modifié.

Exercice 1 - 5 points.

On considère un tableau TAB constitué de n entiers positifs.
Un tableau est dit **monotone** si la séquence des éléments consécutifs du tableau est monotone (croissante ou décroissante).
Un tableau est **unimodal** s'il est consitué d'exactement deux séquences monotones.
Exemples: 1 — 5 — 6 — 10 — 14 — 20 est monotone. 1 — 5 — 14 — 20 — 10 — 6 est unimodal. 1 — 10 — 6 — 5 — 14 — 20 n'est ni monotone ni unimodal.

a) Écrire et analyser un algorithme qui vérifie qu'un tableau est monotone (resp. unimodal). [1,5 points par algorithme]

b) Il s'agit maintenant d'écrire et d'analyser un algorithme de complexité $O(n \lg n)$ qui, partant d'un tableau quelconque, le réorganise en un tableau unimodal. [2 points]

Exercice 2 (Arbres) - 6,5 points.

a) [4 points] Construire par insertions successives de $3, 5, 1, 6, 8, 11, 2$, en précisant les arbres obtenus à chaque étape :

- le tas correspondant [1,5 points]
- l'ABR correspondant [0,5 points]
- l'ARN correspondant [2 points]

b) [1,5 points] On rappelle qu'un arbre binaire de hauteur h est dit *complet* si pour tout $p \leq h$ il possède exactement 2^p noeuds de profondeur p. Dessinez un arbre complet dont les noeuds contiennent les valeurs $1, 2, 3, 4, 5, 6, 7$ et qui soit un ABR. Effectuez une rotation gauche sur la racine, puis une rotation droite sur le noeud étiqueté par 4. Donnez alors une coloration pour obtenir un arbre rouge noir.

c) [1 point] Quel est le nombre maximum de noeuds dans un ARN de hauteur noire h ?

Problème 3 - 10,5 points.

Supposez d'avoir une liste de n éléments, chacun avec une clé qui est un entier compris entre 1 et k. Le principe du *bucket sort* (en français, bucket = seau) est de distribuer les n éléments en k

"seaux" (un seau pour chaque valeur de la clé) et ensuite réunifier les seaux pour retourner une liste triée.

a) [3 points] Donner le pseudo-code d'un algorithme basé sur le *bucket sort* qui trie une liste chaînée (c-à-d, retourne la liste chaînée triée), tel que la complexité en temps de l'opération de distribution dans les seaux ne dépasse pas $O(n)$ et la complexité en temps de l'opération de réunion des seaux ne dépasse pas $O(k)$.
Pour résoudre le problème, vous pouvez utiliser des structures auxiliaires que vous jugerez opportunes, sans dépasser la limite d'une taille $O(n+k)$. Dans ce cas, vous devrez illustrer les structures que vous avez utilisées et donner la complexité en espace aussi précisément que possible.
Un algorithme de tri est dit *stable* s'il préserve l'ordre relatif des éléments ayant clés identiques (si x précède y avant l'exécution du tri et $clé(x) = clé(y)$, alors x précède y après l'exécution du tri). Votre algorithme doit etre stable et vous donnerez une preuve ce fait.

b) [0,5 points] Quel est le temps nécessaire pour trier 1.000.000 de personnes par date d'anniversaire (uniquement jour et mois) avec votre algorithme de *bucket sort*?

c) [0,5 points] Pour quelles valeurs de k relativement à n cet algorithme est-il plus intéressant? A quoi peut-on assimiler $O(n+k)$ dans ce cas?

La suite du problème montre une solution possible pour les cas des valeurs de k et n pour lesquelles *bucket sort* est moins intéressant.

Le *radix sort* effectue une série de *bucket sorts* successifs. Bien que ceci ne soit pas intuitif, dans la première exécution de *bucket sort* on utilise comme clé le chiffre le moins significatif de la clé d'origine (celui des unités). Dans la deuxième exécution de *bucket sort* on utilise comme clé le deuxième chiffre le moins significatif de la clé d'origine (celui des dizaines, si la base est 10), et ainsi de suite, jusqu'à la dernière exécution de *bucket sort*, où on utilise comme clé le chiffre le plus significatif de la clé d'origine.

d) [1 point] Simuler le comportement de *radix sort* sur la liste $253, 346, 1034, 10, 5, 127, 218$.

e) [1,5 points] Donner le pseudo-code d'un algorithme basé sur *radix sort* et qui, de préférence, utilise votre algorithme pour *bucket sort* comme sous-module.

f) [1,5 points] Donner une preuve que *radix sort* trie correctement une liste.

g) [0,5 points] Les exécutions de *bucket sort* appelées par *radix sort* sont-elles dans le cas plus intéressant ou moins intéressant?

h) [0,5 points] Combien d'exécutions de *bucket sort* appelle-t-il *radix sort*?

i) [0,5 points] Quelle est donc la complexité en temps de *radix sort*?

j) [1 point] Pour quelles valeurs de k relativement à n *radix sort* est-il asymptotiquement plus rapide que *quick sort* ?