IFT2015 A13 :: Examen final[1]

[1] English translation at the end.

Miklós Csűrös

13 décembre 2013

AUCUNE DOCUMENTATION n'est permise. L'examen vaut 150 points, et vous pouvez avoir jusqu'à 25 points de boni additionnels. Décrivez vos algorithmes en pseudocode ou en Java(-esque).

► **Répondez à toutes les questions dans les cahiers d'examen.**

F0 Votre nom (1 point)

► Écrivez votre nom et code permanent sur tous les cahiers soumis.

F1 Table de symboles (39 points)

(a) 3 structures, 3 opérations, 3 performances (27 points) On a vu plusieurs structures de données qui peuvent servir à implémenter le type abstrait de la table de symboles. L'efficacité des implémentations n'est pas la même : ici vous devez comparer le temps de calcul pour trois opérations fondamentales : (1) insertion, (2) recherche fructueuse, et (3) recherche infructueuse. ► Donnez le temps de calcul des trois opérations avec les trois structures de données suivantes : (A) liste chaînée non-triée, (B) arbre rouge-et-noir, et (C) tableau de hachage avec sondage linéaire, dont la facteur de remplissage $\alpha < 0.5$. Spécifiez le temps de calcul comme une fonction du nombre des éléments n, en utilisant la notation asymptotique, dans trois cas : (i) le pire cas, (ii) le meilleur cas, et (iii) en moyenne. Il ne faut pas justifier vos réponses.

(b) Justifications (12 points) ► Élaborez *3* de vos réponses (votre choix lesquels parmi les 27) en **(a)** : expliquez comment la structure assure un tel temps de calcul.

F2 Tri rapide sans récursion (25+5 points)

On décrit le tri rapide souvent comme un algorithme récursif :

```
    Algo QUICKSORT(A,l,r)                    // sorts A[l..r−1]
1.  if r − l ≤ 1 then return
2.  i ← PARTITION(A,l,r)   // partitioning with a pivot at i
3.  QUICKSORT(A,l,i−1)
4.  QUICKSORT(A,i+1,r)
```

(a) Tri rapide avec une pile (15 points). ► Donner une implantation du tri rapide sans récursion, en utilisant une pile explicitement. Il n'est pas nécessaire d'implanter PARTITION().

Indice: Empiler/dépiler les indices (l,r) des sous-tableaux dans une boucle.

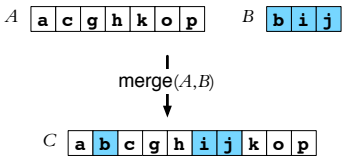
(b) Tri rapide avec une queue (10 points). Est-ce que l'algorithme de **(a)** s'éxécute correctement si on utilise une queue FIFO au lieu de la pile ? ►Justifiez votre réponse.

(c) Tri rapide avec une petite pile (5 points boni). ♡ Contrôlez la taille maximale de la pile : donnez un algorithme itératif en **(a)** qui utilise $O(\lg(n))$ mémoire au pire.

Indice: On peut exécuter les lignes 3 et 4 dans n'importe quel ordre — l'un ou l'autre appel peut être en position terminale ce qu'on peut transformer à une boucle **while**.

F3 Fusion optimale (25 points)

Supposons qu'on veut fusionner deux tableaux $A[0..n-1]$ et $B[0..m-1]$ triés dans l'ordre croissant des éléments. La fusion crée un troisième tableau qui contient tous les éléments de A et B, dans l'ordre croissant. L'algorithme a le droit de comparer un élément $A[i]$ à un autre élément $B[j]$ à la fois. Soit $C(n,m)$ le nombre de comparaisons au pire qu'un algorithme utilise pur fusionner des tableaux de tailles $m \leq n$.



(a) Borne inférieure (15 points). ► Démontrer que $C(n,m) \geq \lg\binom{n+m}{n}$ pour tout algorithme de fusion.

(a) Tableaux de même taille (10 points). ► Démonter que la procédure[2] du tri par fusion est très proche à la borne inférieure quand $m = n$. Utilisez la formule de Stirling[3] pour $\binom{2n}{n} = (n!)^{-2} \cdot ((2n)!)$.

[2] Fusion de deux tableaux :

```
Algo MERGE(A[0..n−1], B[0..m−1])
initialize C[0..n+m−1]
i ← 0; j ← 0
for k ← 0, 1, ..., n+m−1 do
    if i = n then C[k] ← B[j]; j ← j+1
    else if j = m then C[k] ← A[i]; i ← i+1
    else if A[i] ≤ B[j] then C[k] ← A[i]; i ← i+1
    else C[k] ← B[j]; j ← j+1
```

[3] Stirling : $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

F4 Longueur du chemin interne (25+10 points)

La *longueur du chemin interne* dans un arbre binaire T se définit comme la somme des profondeurs (niveaux) des nœuds internes :

$$P(T) = \sum_{x \in T} d(x),$$

où $d(x)$ dénote le niveau du nœud interne x.

(a) Calculer la longueur du chemin (25 points). ► Donner un algorithme récursif qui calcule $P(T)$ dans un arbre binaire T.

Indice: Écrire le code pour $\mathsf{pathlength}(x, \mathsf{depth})$ qui calcule et retourne la somme $\sum_y d(y)$ sur les nœuds internes $y \in T_x$ dans le sous-arbre enraciné à x — mettre la profondeur de x comme argument.

(b) Solution itérative (10 points boni). ♡ Donner un algorithme non-récursif qui calcule $P(T)$ en utilisant un espace de travail de $\Theta(1)$.

Indice: Faire un parcours infixe. Écrivez le code pour trouver le successeur d'un nœud sans ou avec enfant droit externe, et modifiez l'algorithme pour suivre les niveaux.

F5 Sondage linéaire (35+10 points)

Dans un tableau de hachage avec sondage linéaire, on insère un nouvel élément x dans la première cellule non-occupée dans la séquence $T[i], T[(i+1) \bmod M], T[(i+2) \bmod M], \ldots$, à partir de l'indice $i = h(x)$ donné par la fonction de hachage.

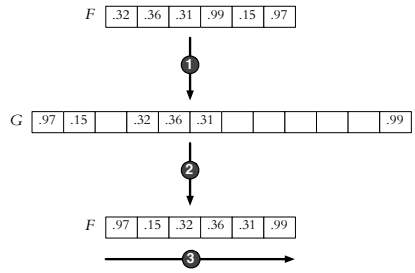
(a) L'ordre n'a pas d'importance (15 points). On se demande s'il est possible de réarranger les éléments dans le tableau de hachage (on reste avec la même fonction de hachage) afin d'améliorer l'efficacité de la recherce. ▶ Démontrer que la recherche fructueuse a le même coût[4] moyen après l'insertion des éléments $\{x_1, x_2, \ldots, x_n\}$ dans n'importe quel ordre.

Hint: Considérez comment on change le coût de la recherche si on insère dans l'ordre $x_1, x_2, \ldots, x_{i+1}, x_i, \ldots, x_n$ au lieu de $x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_n$, et argumentez qu'on peut produire toute permutation par échanges de voisins.

[4] coût = nombre de cases examinées dans le tableau

(b) Tri par hachage (20 points). Supposons qu'on veut trier un tableau $F[0..n-1]$ de nombres flottants $0 \leq F[i] < 1$. Considérez la démarche suivante : (0) Créer un tableau $G[0..2n-1]$,avec cellules vides[5] au début. (1) Utiliser G comme un tableau avec la fonction de hachage $h(x) = \lfloor 2n \cdot x \rfloor \in \{0, 1, \ldots, 2n-1\}$: insérer les $F[i]$ un-à-un avec sondage linéaire. (2) Copier le contenu des cases occupées de G à F dans le même ordre, et (3) trier F par insertion.

▶ Donner le code pour un tel algorithme avec tous les détails (incluant tri par insertion et insertion dans le tableau de hachage ; on peut même combiner (2)+(3) dans une seule boucle).



[5] En Java, on peut mettre `G[i]=Double.NaN;`

(c) Temps linéaire. (10 points boni). ♡ Démontrer que l'algorithme de **(b)** prend $\Theta(n)$ temps à la moyenne (assumer que les $F[i]$ sont uniformement distribués).

BONNE CHANCE !

English translation

NO DOCUMENTATION is allowed. The examen is worth 150 points, and you can collect an additional 25 bonus points. Describe your algorithms in pseudocode or Java(-esque).

Answer each question in the exam booklet.

E0 Your name (1 point)

► Write your name and *code permanent* on each booklet that you submit.

E1 Symbol table (39 points)

(a) 3 structures, 3 operations, 3 performance guarantees (27 points) We have seen a number of data structures that can be used to implement the abstract data type of symbol table. Not all implementations have the same performance. You need to compare the running times for three fundamental operations : (1) insertion, (2) successful search and (3) unsuccessful search.
► Give the running times for the three operations in the following three data structures : (A) unsorted linked list, (B) red-black tree, and (C) hash table with linear probing and a load factor of $\alpha < 0.5$. Give the (i) worst-case, (ii) best-case and (iii) average-case running times in asymptotic notation as a function of the number n of elements. (That is, 27 statements about time complexity.) You do not need to justify your answers.

(b) Justifications (12 points) ► Develop *3* of your answers (of your choice among the 27) from *(a)*. Explain how the structure allows for such performance.

E2 Quicksort without recursion (25+5 points)

Quicksort is usually described as a recursive procedure :

```
   Algo QUICKSORT(A,l,r)                    // sorts A[l..r−1]
1. if r − l ≤ 1 then return
2. i ← PARTITION(A,l,r)   // partitioning with a pivot at i
3. QUICKSORT(A,l,i−1)
4. QUICKSORT(A,i+1,r)
```

(a) Quicksort with a stack (15 points). ► Give a non-recursive implementation of quicksort using a stack explicitly. You do not need to implement PARTITION().

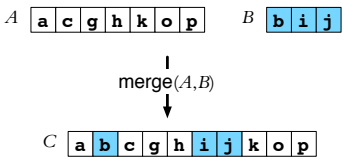
Hint: Push/pop the subarray indices (l,r) within a loop.

(b) Quicksort with a queue (10 points). Would the algorithm work correctly if you used a FIFO queue instead of a stack in *(a)*? ► Explain your answer.

(c) Quicksort with a small stack (5 bonus points). ♡ Control the maximum stack depth : give an iterative algorithm in *(a)* with guaranteed $O(\lg(n))$ memory use.

Hint: Lines 3 and 4 can be executed in any order — either of them can be in terminal position which can be converted into a **while** loop.

E3 Optimal merging (25 points)



Suppose that one wants to merge two tables $A[0..n-1]$ et $B[0..m-1]$, each sorted in increasing order of keys. That is, the merge should produce a third table $C[0..n+m-1]$ that contains all elements of A and B in sorted order. The algorithm can compare one element $A[i]$ to another element $B[j]$ at a time : we want to minimize the number of comparisons in the worst-case. Let $C(n,m)$ denote the worst-case number of comparisons made by a given algorithm for input arrays of size $m \leq n$.

(a) Lower bound (15 points). ► Prove that $C(n,m) \geq \lg \binom{n+m}{n}$ for all merging algorithms.

(a) Optimal for equal length (10 points). Show that mergesort's procedure[6] nearly matches the lower bound when $m = n$ by using Stirling's approximation[7] for $\binom{2n}{n} = (n!)^{-2} \cdot ((2n)!)$.

[6] Merging in mergesort :

```
Algo MERGE(A[0..n−1], B[0..m−1])
initialize C[0..n+m−1]
i ← 0; j ← 0
for k ← 0,1,...,n+m−1 do
    if i = n then C[k] ← B[j]; j ← j+1
    else if j = m then C[k] ← A[i]; i ← i+1
    else if A[i] ≤ B[j] then C[k] ← A[i]; i ← i+1
    else C[k] ← B[j]; j ← j+1
```

[7] Stirling : $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

E4 Path length in a tree (25+10 points)

The (internal) *path length* in a binary tree T is defined as the sum of the depths of internal nodes :

$$P(T) = \sum_{x \in T} d(x),$$

where $d(x)$ is the depth of internal node x.

(a) Computing path length (25 points). ► Give a recursive algorithm that computes $P(T)$ for a binary tree T.

Hint: Write the code for $\mathsf{pathlength}(x, \mathsf{depth})$ that computes and returns the sum $\sum_y d(y)$ over the internal nodes $y \in T_x$ in the subtree rooted at node x — give the depth of x as an argument.

(b) Iterative solution (10 bonus points). ♡ Give a non-recursive algorithm that computes $\sum_{x \in T} d(x)$ over the internal nodes using $\Theta(1)$ work space.

Hint: Do an infix traversal. Write the code for finding the successor of a node with or without right child, and modify it to keep track of the depth.

E5 Linear probing (35+10 points)

In a hashtable with linear probing, one inserts a new element x at the first empty cell found in the sequence $T[i], T[(i+1) \bmod M], T[(i+2) \bmod M], \ldots$ starting with $i = h(x)$ that x hashes to.

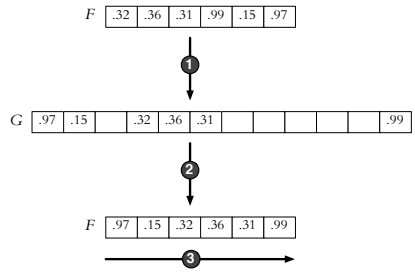
(a) Order does not matter (15 points). One may wonder if rearranging the elements in the table (while keeping the same hash function) would improve the search. ► Prove that inserting the elements $\{x_1, x_2, \ldots, x_n\}$ *in any order* into $T[0..M-1]$ with $M > n$ results in the same average cost[8] for successful search.

Hint: Consider how the search cost changes if you insert in the order $x_1, x_2, \ldots, x_{i+1}, x_i, \ldots, x_n$ instead of $x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_n$, and argue that any order can be constructed by swaps of adjacent elements.

[8] cost = number of cells accessed in the table

(b) Sorting by hashing (20 points). Suppose that you want to sort an array $F[0..n-1]$ of floating-point numbers $0 \leq F[i] < 1$. Consider the following algorithm : (0) Create an array $G[0..2n-1]$, with empty cells[9] initially. (1) Use G as a hash table with the hash function $h(x) = \lfloor 2n \cdot x \rfloor \in \{0, 1, \ldots, 2n-1\}$: insert $F[i]$ into G one by one with linear probing. (2) Copy the non-empty elements of G back into F, preserving their order, and (3) sort F by insertion sort.

► Give the pseudocode for the algorithm *with all details* (including insertion sort and hash table insertion ; you may even combine (2) and (3) into a single loop).



[9] In Java you can put `G[i]=Double.NaN;`

(c) Linear time. (10 bonus points). ♡ Prove that the algorithm's expected running time from (b) (assuming $F[i]$ are uniformly distributed) is $\Theta(n)$.

GOOD LUCK !