

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

Corrigé de la série 6

26 Octobre 2007

1. Diviser pour régner : Le problème du skyline

a) Nous parcourons l'axe des x de gauche à droite, en regardant les points un par un. Pour chaque point de L et de R nous devons décider s'il faut ajouter un point C , et si oui quelle va être sa hauteur.

Un algorithme est donnée ci-dessous. On remarque que le nombre d'opérations est $O(|L| + |R|)$ où $|L|$ et $|R|$ représentent les nombres de points dans les skylines de L et R .

Dans l'algorithme ci-dessous, les skylines sont données sous forme de tableau, avec les entrées étant des couples (x, h) . Par convention, la dernière entrée est toujours $(\infty, 0)$ et marque la fin de la skyline.

Call : SKYLINEMERGE

Input: Skylines L et R

Output: Skyline ret , superposition des deux skylines.

```

 $i_\ell, i_r \leftarrow 0$ 
 $h_\ell \leftarrow 0, h_r \leftarrow 0, h \leftarrow 0$ 
while  $\min(L[i_\ell].x, R[i_r].x) < \infty$  do
  if  $L[i_\ell].x < R[i_r].x$  then
     $h_\ell \leftarrow L[i_\ell].h$ 
     $x \leftarrow L[i_\ell].x$ 
     $i_\ell \leftarrow i_\ell + 1$ 
  else
     $h_r \leftarrow R[i_r].h$ 
     $x \leftarrow R[i_r].x$ 
     $i_r \leftarrow i_r + 1$ 
  if  $\max(h_\ell, h_r) \neq h$  then
     $h \leftarrow \max(h_\ell, h_r)$ 
     $ret.add((x, h))$ 
 $ret.add((\infty, 0))$ 
return  $ret$ 

```

Dans le pseudocode ci-dessus, la méthode `add` ajoute un élément à la fin d'un array.

b) Remarquons d'abord que l'algorithme SKYLINEMERGE ne résout pas le problème SKYLINE. En effet, SKYLINE attend comme input un ensemble de bâtiments (i.e. de triplets (x_1, x_2, h)) et retourne comme output le skyline correspondant. Par contre, SKYLINEMERGE a comme input deux skylines et retourne le skyline obtenu en les combinant.

MergeSort : Notre solution est très similaire à *MergeSort*, nous rappelons donc les détails de cet algorithme. Le problème à résoudre est d'ordonner une suite de nombres naturels (l'input est donc une suite de nombres naturels, et l'output une suite qui contient les mêmes éléments, mais ordonnés). Nous avons d'abord construit un algorithme MERGE qui, étant donné deux suites S_1 et S_2 de nombres naturels *ordonnées*, retourne une suite ordonnée contenant les éléments de S_1 et S_2 .

L'algorithme MERGESORT divise la suite donnée en 2 sous-suites (plus petites), qui sont ensuite ordonnées (avec un appel récursif à MERGESORT). Puis, ces 2 sous-suites ordonnées sont combinées avec l'algorithme MERGE pour obtenir une version ordonnée de la suite de l'input.

Puisque nous avons fait un appel récursif à MERGESORT, chacune des sous-suites sera elle-même divisée en deux sous-suites, etc. Il faut donc définir un cas de base où la récurrence se terminera. Il s'agit du cas où la suite ne contient qu'un seul élément et est donc forcément déjà ordonnée.

On l'appelle un algorithme diviser-pour-régner puisqu'on divise le problème en deux instances plus petites du même problème. l'algorithme MERGE est utilisé pour recoller les solutions de ces instances pour obtenir la solution du problème original.

Pour résoudre le problème SKYLINE nous procédons de la même façon. Notre algorithme (que nous appelons SKYLINE SOLVE) commence par diviser la liste de bâtiments en deux sous-listes (dont la taille vaut à peu près la moitié de celle de la suite originale). Nous trouvons ensuite le Skyline de chacune de ces suites avec un appel récursif à SKYLINE SOLVE. Nous réduisons donc notre problème à deux sous-problèmes qui sont toujours des instances de SKYLINE, mais avec un input plus petit (il y a moins de bâtiments).

Les solutions de ces deux sous-problèmes (les deux skylines) sont ensuite combinées avec l'algorithme SKYLINE MERGE donné dans la partie a) (ce pas correspond donc à MERGE). Comme avec MERGESORT, puisqu'il y a cet appel récursif, il faut définir un cas de base. C'est le cas où il n'y a qu'un seul bâtiment, et le skyline est égal à ce bâtiment.

On obtient au final, en pseudocode :

Call : SKYLINE SOLVE

Input: *Bat* : Une liste de n bâtiments (de triplets (x_1, x_2, h))

Output: *A* : Le skyline correspondant (une suite de points (x, h))

if $n = 1$ **then**

return $((Bat[1].x_1, Bat[1].h), (Bat[1].x_2, 0))$

else

$m \leftarrow \lceil n/2 \rceil$

*Bat*₁ \leftarrow la sous-suite de *Bat* composée des $m - 1$ premiers éléments de *Bat*

*Bat*₂ \leftarrow la sous-suite de *Bat* composée des autres éléments (bâtiments) de *Bat*

*Skyline*₁ \leftarrow SKYLINE SOLVE(*Bat*₁)

*Skyline*₂ \leftarrow SKYLINE SOLVE(*Bat*₂)

return SKYLINE MERGE(*Skyline*₁, *Skyline*₂)

2. Algorithmes

a) Le input consiste en une suite triée d'éléments de \mathbb{N} et un élément de \mathbb{N} . Nous avons donc :

$$I = \left\{ ((a_0, \dots, a_{n-1}), x) \in \mathbb{N}^* \times \mathbb{N} \mid \forall i \in \{0, \dots, n-2\} : a_i \leq a_{i+1} \right\}.$$

Il s'agit d'un problème de décision, ainsi

$$O = \{vrai, faux\}.$$

Pour exprimer formellement le fait qu'il existe deux éléments distincts d'une suite (a_0, \dots, a_{n-1}) dont la somme vaut x nous écrivons :

$$\exists i, j \in \{0, \dots, n-1\} : i \neq j \wedge a_i + a_j = x.$$

Ainsi nous pouvons déduire :

$$R = \left\{ \left((a_0, \dots, a_{n-1}), x \right), vrai \right\} \in I \times O \mid \exists i, j \in \{0, \dots, n-1\} : i \neq j \wedge a_i + a_j = x \right\} \\ \cup \left\{ \left((a_0, \dots, a_{n-1}), x \right), faux \right\} \in I \times O \mid \nexists i, j \in \{0, \dots, n-1\} : i \neq j \wedge a_i + a_j = x \right\}$$

- b) L'algorithme naïf consisterait à considérer une par une toutes les paires d'éléments de la suite, et vérifier si leur somme vaut x . Puisqu'il y a $O(n^2)$ telles paires, le temps de parcours de cet algorithme est $O(n^2)$.
- c) L'algorithme naïf ci-dessus n'utilise pas le fait que la suite est *triée*.

Informellement, nous commençons avec a_0 (le plus petit élément) et a_{n-1} (le plus grand élément).

- Si $a_0 + a_{n-1} = x$ nous savons que le output est *vrai*.
- Si $a_0 + a_{n-1} < x$ alors il n'y aura aucune solution utilisant a_0 (même avec l'élément le plus grand, la somme reste inférieure à x). Nous pouvons donc oublier l'élément a_0 et continuer avec la suite (a_1, \dots, a_{n-1}) .
- Si $a_0 + a_{n-1} > x$ alors par le même raisonnement que ci-dessus nous pouvons éliminer a_{n-1} et continuer avec (a_0, \dots, a_{n-2}) .

Une implémentation est donnée ci-dessous.

```

Call : ALGO( $(a_0, \dots, a_{n-1}), x$ )
1:  $i \leftarrow 0$ 
2:  $j \leftarrow n - 1$ 
3: while  $i \neq j$  do
4:   if  $a_i + a_j = x$  then
5:     return TRUE
6:   else if  $a_i + a_j < x$  then
7:      $i \leftarrow i + 1$ 
8:   else
9:      $j \leftarrow j - 1$ 
10: return FALSE
  
```

Nous voyons qu'à chaque itération de la boucle **while** (lignes 3–9), soit i est incrémenté, soit j est décrémenté. Ainsi cette boucle sera exécutée au plus $n - 1$ fois. Puisque toutes les opérations à l'intérieur de cette boucle prennent un temps constant, l'algorithme est bien $O(n)$.

3. Algorithmes par induction : Le problème de la célébrité

On précise que dans nos graphes non-orientés on suppose qu'un sommet n'est jamais connecté à lui-même, c'est-à-dire que la diagonale de la matrice d'adjacence ne contient que des 0.

- a) Le sommet 2 est une célébrité puisqu'il est connu de tout le monde mais ne connaît personne (son in-degree vaut $n - 1$ alors que son out-degree vaut 0).
- b) Supposons que nous avons un graphe avec deux célébrités (que nous appelons les sommets i et j). i ne connaît personne, donc en particulier i ne connaît pas j . j ne peut donc pas être une célébrité et nous avons une contradiction. Un graphe ne peut donc pas avoir deux célébrités.
- c) On rappelle que pour tout graphe orienté, avec comme matrice d'adjacence A , A_{ij} vaut 1 s'il y a un arc du sommet i au sommet j , et 0 sinon. Ainsi la ligne k montre les arcs qui sortent du sommet k , et la colonne k ceux qui vont vers k .

Si le sommet k est une célébrité, la ligne k ne contient donc que des 0. De même, la colonne k ne contient que des 1, sauf sur la diagonale (l'entrée A_{kk}) qui vaut 0 (puisque nous supposons qu'il n'y a jamais d'arc d'un sommet à lui-même).

- d) L'algorithme naïf serait de prendre chaque sommet un par un et de regarder s'il s'agit d'une célébrité. Etant donné un sommet k , pour déterminer si c'est une célébrité nous devons :

- (1) vérifier que tous les autres sommets connaissent k
- (2) vérifier qu'aucun autre sommet n'est connu de k

Pour (1) nous devons vérifier que $A_{ik} = 1$ pour tout $i \in \{1, \dots, n\} \setminus \{k\}$ ($n - 1$ questions).

Pour (2) nous devons vérifier que $A_{kj} = 0$ pour tout $j \in \{1, \dots, n\} \setminus \{k\}$ ($n - 1$ questions).

Pour vérifier si k est une célébrité il nous faut donc $2 \cdot (n - 1)$ questions.

Puisque nous devons répéter cette procédure pour tous les $k \in \{1, \dots, n\}$ on obtient un total de $2 \cdot n \cdot (n - 1)$ questions dans le pire des cas, donc un algorithme $O(n^2)$.

- e) Rappelons qu'on sait qu'il existe une célébrité dans le graphe qui nous est donné.
- (i) Si on a deux sommets a et b on peut poser la question "a connaît-il b?". Si la réponse est "oui" on sait que a ne peut pas être une célébrité, et puisque l'un des deux doit être une célébrité, il s'agit forcément de b . De même, si la réponse est "non", on sait que b ne peut pas être une célébrité, et donc la célébrité est a .
 - (ii) Nous supposons qu'étant donné un groupe de n personnes contenant une célébrité, nous pouvons la retrouver en posant $T(n)$ questions.
Etant donné un groupe de $n + 1$ personnes nous procédons comme suit :
 - nous choisissons deux personnes a et b parmi les $n + 1$.
 - nous posons la question "a connaît-il b?"
 - Si la réponse est "oui" nous savons que a n'est pas la célébrité. Celle-ci se trouve donc parmi les n autres personnes (dont b). Nous pouvons donc la retrouver en posant $T(n)$ questions. Nous avons donc dû poser $T(n) + 1$ questions au total.
 - Si la réponse est "non" nous savons que b n'est pas la célébrité. Nous pouvons donc de la même façon la retrouver parmi les n personnes restantes en posant $T(n)$ questions. Nous avons donc dû poser $T(n) + 1$ questions au total.
 - (iii) L'algorithme récursif a été décrit dans la partie (ii). En pseudo-code on obtient :

Call : FINDCELEBRITY

Input: $S = \{s_1, \dots, s_n\}$: Un ensemble de n personnes, dont une célébrité

Output: c : la célébrité

```
if  $n = 2$  then
  if  $s_1$  connaît  $s_2$  then
    return  $s_2$ 
  else
    return  $s_1$ 
if  $s_1$  connaît  $s_2$  then
  return FINDCELEBRITY( $S \setminus \{s_1\}$ )
else
  return FINDCELEBRITY( $S \setminus \{s_2\}$ )
```

Si on appelle $T(n)$ le nombre de questions dont il a besoin pour trouver la célébrité parmi n personnes on a donc $T(2) = 1$ et $T(n) = T(n - 1) + 1$ pour $n \geq 2$. Il est assez clair que cela nous donne

$$T(n) = n - 1$$

En effet par récurrence on voit que l'égalité est vraie pour $n = 2$ (base), et si elle est vraie pour n alors

$$T(n + 1) = T(n) + 1 = (n - 1) + 1 = n$$

Ainsi c'est un algorithme $O(n)$.