ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Section d'Informatique et de Systèmes de Communication

Corrigé de la série 4 12 octobre 2007

1. Exponention Rapide

a) Comme $a_j = a_{k_j} + a_{l_j}$, on a

$$x^{a_j} = x^{a_{k_j}+a_{l_j}} = x^{a_{k_j}} x^{a_{l_j}}. \tag{1}$$

Donc si, pour un $0 \leq j \leq r$ fixé, $x^{a_0}, \ldots, x^{a_{j-1}}$ sont tous connus, on peut calculer x^{a_j} en une multiplication en utilisant (1).

Donc, comme $x^{a_0} = x^1 = x$, nous pouvons conclure par induction qu'on peut avec r multiplications calculer x^n en se servant de la chaîne d'addition.

Notons que donc une chaîne d'addition nous fournit une recette comment calculer une puissance x^n. Une chaîne courte correspond à une méthode n'utilisant que très peu de multiplications.

b) L'idée est d'écrire $x^{nm} = (x^n)^m$, et de calculer $y := x^n$ en $\ell(n)$ multiplications et y^m en $\ell(m)$ multiplications. Comme $y^m = x^{nm}$, cela suffit pour montrer le résultat, si on peut faire ce même calcul avec les chaînes d'additions.

Soit $a_0, \ldots, a_r$ une chaîne d'addition pour m telle que $r = \ell(m)$, et soit $b_0 = 1, b_1, b_2 \ldots, b_s$ une chaîne d'addition pour n, avec $s = \ell(n)$. Ecrivons

$$b_j = b_{l_j} + b_{k_j}.$$

Alors si je pose $c_0 = w$ pour un w quelconque, et

$$c_j := c_{l_j} + c_{k_j}, \qquad j = 1, \ldots, s,$$

on a par induction que $c_j = wb_j$ pour tout $j = 1, \ldots, s$. En particulier, je peux choisir $w = a_r$, et ensuite la chaîne

$$a_0, a_1, \ldots, a_{r-1}, a_r = c_0, c_1, \ldots, c_s = nm$$

est une chaîne d'addition de longueur $r + 1 + s$, et donc correspond à un algorithme pour calculer x^{nm} en $r + s = \ell(m) + \ell(n)$ étapes.

c) Remarquons d'abord qu'on peut enlever sans autre tous les éléments $> n$ de la chaîne d'addition, et ça reste une chaîne d'addition (car les a_i sont non-négatifs). De même, on peut enlever les duplicats.

On peut finalement ordonner les éléments, car la règle de définition respecte l'ordre, i.e., si $a_k + a_l = a_j$, alors $a_j > a_k, a_l$ et donc je peux certainement calculer a_j avec une addition valide si toutes les éléments inférieures à a_j précèdent a_j.

d) On vient de voir qu'on peut supposer la chaîne optimale croissante. On a $a_j = a_{k_j} + a_{l_j}$ avec $k_j, k_l \leq j - 1$ et donc on peut borner

$$a_j = a_{k_j} + a_{l_j} \leq a_{j-1} + a_{j-1} = 2a_{j-1},$$

ce qui permet de conclure que $a_j \leq 2^j$, en utilisant de plus que $a_0 = 1$.

Donc, en particulier, si $a_0, \ldots, a_r$ est une chaîne d'addition pour calculer n, on a $n = a_r \leq 2^r$, ou encore $\log_2(n) \leq r$. Le résultat suit en prenant une chaine de longueur $r := \ell(n)$.

2. Stacks et files d'attente

a) Soient S_1 et S_2 deux stacks. Nous aimerions réaliser une file d'attente Q en utilisant S_1 et S_2.

Nous pouvons réaliser une telle file d'attente qui contient à $top[S_1]$ l'élément le plus récemment ajouté, et la file continue alors jusqu'à la fin de S_1, continue à la fin de S_2 jusqu'à $top[S_2]$ qui contient l'élément le plus ancien.

L'implémentation de `Enqueue`(x) est triviale :

1: `Push`(S_1, x)

L'idée de l'algorithme pour `Dequeue`(x) est d'enlever un élément de S_2. S'il n'y en a plus, on bouge le contenu de S_1 dans S_2 et on ressaye. Le voici :

1: $r \leftarrow$ `Pop`(S_2)
2: **if** $r =$ underflow **then**
3: $\quad r \leftarrow$ `Pop`(S_1)
4: $\quad$ **while** $r \neq$ underflow **do**
5: $\quad\quad$ `Push`(S_2, r)
6: $\quad\quad r \leftarrow$ `Pop`(S_1)
7: $\quad r \leftarrow$ `Pop`(S_2)
8: **return** r

b) Avec les algorithmes comme présentés sous le point a) ci-dessus, chaque élément est enlevé au plus deux fois est inséré au plus deux fois d'un stack, ce qui fait que la propriété voulue est vérifiée.

3. Arbres

a) On obtient les suites suivantes pour les différents parcours :

Parcours	Suite
Preorder	I, D, A, C, B, G, E, F, H, O, M, K, J, L, N
Inorder	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O
Postorder	B, C, A, F, E, H, G, D, J, L, K, N, M, O, I

b) Il s'agit de voir que pour chaque sommet r la propriété $g < r < d$ est vérifié pour tout sommet g dans le sous-arbre gauche et d dans le sous-arbre droit. Ceci est équivalent à voir que la suite obtenue par le parcours inorder est croissante (cf. point suivant). Ceci est manifestement vrai.

c) " $\Longleftarrow$ ". Nous montrons qu'un arbre de recherche résulte en une suite croissante si parcouru inorder.

Nous prouvons ce résultat par induction (forte) sur les arbres à n sommets. L'arbre vide ($n = 0$), correspond à la suite vide, qui est clairement croissante. Soit maintenant T un arbre avec racine x, sous-arbre gauche L et sous-arbre droit R. La suite obtenue par parcours inorder est

$$[\text{suite du parcours de } L], x, [\text{suite du parcours de } R].$$

Par hypothèse d'induction, la partie L est croissante, et la partie R aussi. Il suffit donc de montrer que x est supérieur à son prédécesseur et inférieur à son succésseur. Comme T

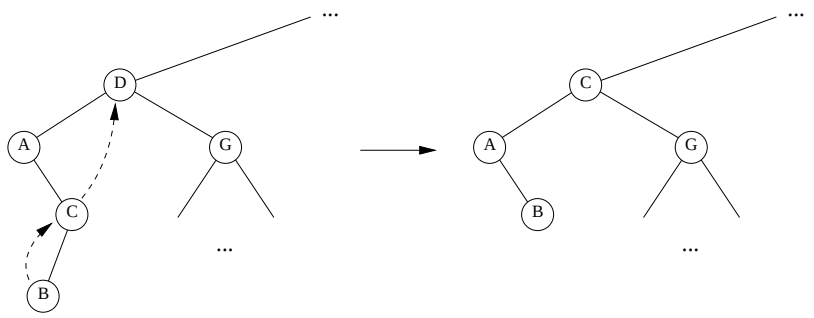
est un arbre de recherche, n'importe quel élément dans L est inférieur à x, donc aussi le dernier de la suite. De manière analogue, il n'y a pas non plus d'inversion entre x et son successeur.

" $\Longrightarrow$ ". Nous montrons que si le parcours inorder est croissant, alors l'arbre en question est un arbre de recherche. Soit x n'importe quel sommet dans l'arbre. Nous montrons que l'opération FIND(x) retourne bien le sommet x, ce qui permet de conclure.

Si x est racine de l'arbre de recherche, il sera clairement trouvé. Sinon, soit z n'importe quel sommet sur le chemin de la racine à x. Supposons sans perdre de généralité que le chemin descend dans le sous-arbre gauche de z, l'autre cas étant analogue. Alors x est parcouru avant z dans la traversée inorder, et donc $x < z$ par la croissance de la suite. Donc à l'étape où FIND se trouve en z, FIND descend aussi dans le sous-arbre gauche.

Comme cet argument s'applique à n'importe quel z sur le chemin vers x, FIND finit par trouver x.

d) On utilise l'algorithme du cours, i.e. on cherche dans le sous-arbre de gauche du sommet D le plus grand membre et on trouve C. Ce sommet ne peut pas avoir de sous-arbre droit (car ceci contredirait sa maximalité), ensuite, on remplace D par C et l'ancien sous-arbre gauche de C est mis à l'ancienne place de C. Schématiquement :



e) L'algorithme effectue plusieurs pas :

[1] Rechercher le sommet x à effacer
[2.0] S'il n'a pas de sous-arbre gauche, alors
Effacer le sommet et mettre son sous-arbre droit à la place. stop.
[2.1] Trouver le plus grand élément du sous-arbre gauche de x :
$y \leftarrow$ `left`$[x]$
Tant que y a un sous-arbre droite
$y \leftarrow$ `right`$[y]$
[2.2] Enlever y de l'arbre et mettre son sous-arbre gauche à sa place
[2.3] Enlever x de l'arbre et mettre y à sa place.

L'algorithme fonctionne parce que le sommet y vérifie la propriété que tous les autres sommets du sous-arbre gauche sont plus petits ; ceci garantit que la propriété de l'arbre de recherche est présérvée. (Voir aussi page 72 des notes de cours).

f) La preuve se fait par induction sur h. Écrivons N_h le nombre maximal de sommets que peut avoir un arbre. Un arbre binaire de hauteur h ne peut avoir qu'un seul sommet (la racine), donc $N_0 = 1 = 2^{0+1} - 1$.

Supposons maintenant l'affirmation prouvée pour h et montrons la pour $h+1$. Notons $\mathcal{T}_h$ l'ensemble d'arbres binaires de hauteur h. Alors comme tout arbre $T \in \mathcal{T}_{h+1}$ est formé

d'une racine et de deux sous-arbres $L, R \in \mathcal{T}_h$ nous avons

$$\begin{aligned} N_{h+1} &= \max_{T \in \mathcal{T}_{h+1}} \{\#(\text{sommets de } T)\} \\ &= \max_{L,R \in \mathcal{T}_h} \{1 + \#(\text{sommets de } L) + \#(\text{sommets de } R)\} \\ &= 1 + \max_{L \in \mathcal{T}_h} \{\#(\text{sommets de } L)\} + \max_{R \in \mathcal{T}_h} \{\#(\text{sommets de } R)\} \\ &= 1 + (2^{h+1} - 1) + (2^{h+1} - 1) \\ &= 2^{h+2} - 1, \end{aligned}$$

ce qui termine la preuve.

4. Jeu d'échecs

a) Pour préparer aux futurs exercices de programmation nous donnons une implémentation en C. Appel `output_valid_posns(n, m, a)` avec nos conventions ci-dessus.

```
void output_valid_posns(int n, int j, int *v)
{
int e;
for(e = 0; e < n; ++e) {
            int i;
            for(i = 0; i < j; ++i) {
                if(v[i] == e || v[i] + i == e + j || v[i] - i == e - j)
                {
break;
                }
            }

    if(i == j) {
     /* No incompatibility found --> placement ok. */
        printf("Can place %dth queen on column %d\n", j + 1, e + 1);
    }
}
}
```

b) L programme suivant C résolut ce problème en utilisant une récursion.

```
#include <stdio.h>
#include <stdlib.h>

/*
    static int is_valid(int n, int j, const int *v)

    Check if the (j + 1)-th positioning of the queen is consistent
    with the previous ones. The proposed positioning is stored in
    v[0], ..., v[j].
 */
```

```
static int is_valid(int n, int j, int *v)
{
    int i;
    for(i = 0; i < j; ++i) {
        if(v[i] == v[j] || v[i] + i == v[j] + j || v[i] - i == v[j] - j)
        {
            return 0;
        }
    }

    return 1;
}

/*
    static void recurse(int n, int j, int *v)

    Output all valid placements of n queens with v[0], ..., v[j - 1]
    being the columns positions of the j first queens. (Queen i is
    always in row i).
 */

static void recurse(int n, int j, int *v)
{
    int i;
    for(i = 0; i < n; ++i) {
        v[j] = i;

        if(is_valid(n, j, v)) {
            if(j == n - 1) {
                printf("Compatible assignment:\n");
                int e;
                for(e = 0; e < n; ++e) {
                    printf("  %d\n", v[e]);
                }
            } else {
                recurse(n, j + 1, v);
            }
        }
    }
}

int main(int argc, char **argv)
{
    int n;
    if(argc != 2 || (n = atoi(argv[1])) <= 0) {
        fprintf(stderr, "usage: size of the chessboard\n");
```

```
            return 1;
        }

        int v[n];
        recurse(n, 0, v);

        return 0;
    }
```

c) Par exemple, dans l'exemple ci-dessus on peut compter le nombre d'appels à `is_valid`. (N.B. Que le temps de parcours de `is_valid` est $O(\text{poly}(n))$.) Par la construction, `is_valid` n'est appelé que pour des j-uples de la forme $v[0] = \sigma(0), v[1] = \sigma(1), \ldots v[j-1] = \sigma(j-1), v[j] = \star$, ou σ est une pérmutation sur $\{0, 1, \ldots, n-1\}$, et $\star$ est une valeur entre 0 et $n-1$ quelconque.

Si l'on fixe σ, en variant j et $\star$, on ne peut obtenir qu'une quantité $O(\text{poly}(n))$ de valeurs possibles, plus précisément $O(n^2)$. Comme il y a $n!$ possibles permutations sur $\{0, \ldots, n-1\}$, on en déduit que `is_valid` est alors appelé $O(n^2 \cdot n!)$ fois.

L'algorithme basique teste n^n configurations, ce qui est beaucoup plus ; le backtracking a donc un avantage substantiel, mais les deux algorithmes sont lents pour de grands n.