ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

Corrigé de la série 6 31 Oct. 2011

1. *Multiplication de plusieurs matrices*

Rappelons que $m_{i,j}$ est le nombre minimal de multiplications scalaires pour le calcul de $A_{i...j}$ et on a:

$$m_{ij} = \begin{cases} 0 & \text{si } i = j = 0, \\ \min_{i \le k < j} \{m_{ik} + m_{k+1,j} + p_{i-1} \cdot p_k \cdot p_j\} & \text{si } 1 \le i \le j \le n. \end{cases}$$

Rappelons aussi que s_{ij} est la valeur de k pour laquelle le minimum de la formule est atteint. Considerons que $M[i,j] = m_{ij}$ et $S[i,j] = s_{i,j}$.

$$\begin{aligned}
m_{1,2} &= 5 \cdot 10 \cdot 3 = 150 \\
m_{2,3} &= 10 \cdot 3 \cdot 12 = 360 \\
m_{3,4} &= 3 \cdot 12 \cdot 5 = 180 \\
m_{1,3} &= \min \begin{cases} m_{1,2} + 5 \cdot 3 \cdot 12 = 330 \\ m_{2,3} + 5 \cdot 10 \cdot 12 = 960 \end{cases} \\
m_{2,4} &= \min \begin{cases} m_{2,3} + 10 \cdot 12 \cdot 5 = 960 \\ m_{3,4} + 10 \cdot 3 \cdot 5 = 330 \end{cases} \\
m_{1,4} &= \min \begin{cases} m_{2,4} + 5 \cdot 10 \cdot 5 = 580 \\ m_{1,2} + m_{3,4} + 5 \cdot 3 \cdot 5 = 405 \\ m_{1,3} + 5 \cdot 12 \cdot 5 = 630 \end{cases}
\end{aligned}$$

Les matrices M et S sont alors:

$$M = \begin{pmatrix} 0 & 150 & 330 & 405 \\ & 0 & 360 & 330 \\ & & 0 & 180 \\ & & & 0 \end{pmatrix} \qquad S = \begin{pmatrix} 0 & 1 & 2 & 2 \\ & 0 & 2 & 2 \\ & & 0 & 3 \\ & & & 0 \end{pmatrix}$$

$S[1,4] = 2$ signifie que $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ est construit de manière optimale à partir de $A_1 \cdot A_2$ et $A_3 \cdot A_4$. Le nombre de multiplications nécessaires est donc 405.

2. *Algorithmes*

a) 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 $O(n^2)$.

b) 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 $(a_1, \ldots, a_{n-1})$.

• Si $a_0 + a_{n-1} > x$ alors par le même raisonnement que ci-dessus nous pouvons éliminer a_{n-1} et continuer avec $(a_0, \ldots, a_{n-2})$.

Une implémentation est donnée ci-dessous.

```
Call: ALGO((a_0, ..., a_{n-1}), x)
 1: i ← 0
 2: j ← n − 1
 3: while i ≠ j do
 4:    if a_i + a_j = x then
 5:       return TRUE
 6:    else if a_i + a_j < x then
 7:       i ← i + 1
 8:    else
 9:       j ← 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 executée au plus $n-1$ fois. Puisque toutes les opération à l'intérieur de cette boucle prennent un temps constant, l'algorithme est bien $O(n)$.

3. Sac à dos 0/1

a) La valeur maximale est 45. Il y a deux solutions correctes: les ensembles qui atteignent ce maximum sont:

$$\{A, C, E\} \qquad \text{et} \qquad \{B, D, E\}$$

Pour les détails voir la figure 1.

Les flèches diagonales corréspondent à la branche passant par le pas 9 de l'algorithme du polycopié et les flèches verticales aux branches passant par 11 ou 14. Les flèches en gris sont les décisions rejetées parce que l'autre chemin est mieux.

Les flèches en gras donnent des chemins optimaux (flèche diagonale à la $i$ème étape: on prend l'objet i; flèche verticale: on ne le prend pas).

b) La solution optimale n'est pas unique, c.f. le corrigé du point précédent. On peut adapter l'algorithme pour trouver toutes les solutions optimales. La méthode la plus simple est d'utiliser une récursion pour le faire (Initialement, nous appélons GETOPTIMALCHOICE2(W, n, C)).

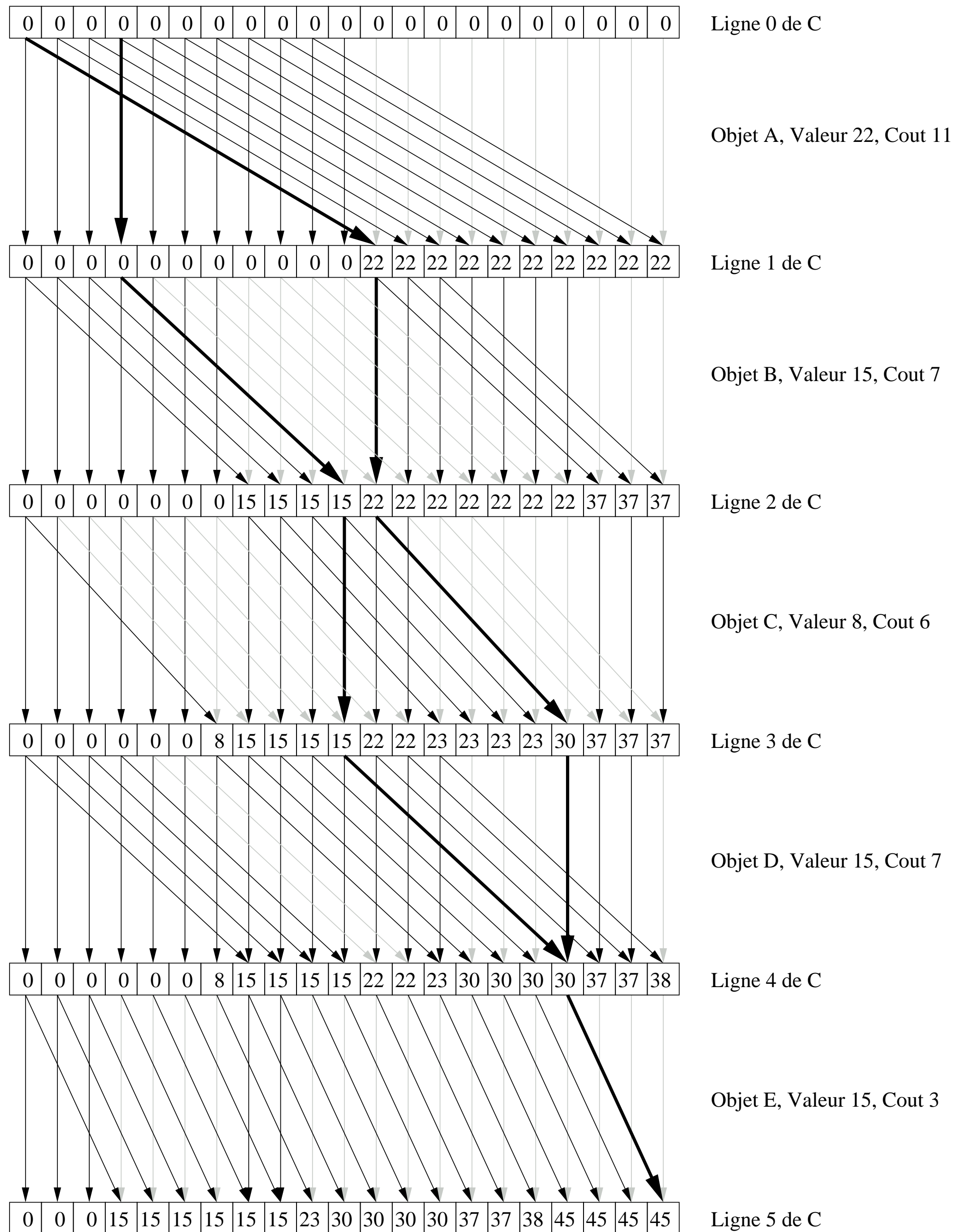

Figure 1: Knapsack 0/1

```
Call: GetOptimalChoice2(w, i, C)
Input: L'input à Knapsack et le C calculé. x[ ] est statique.
Output: Un choix optimal.
  if i ≥ 1 then
    if c_{i-1,w} = c_{i,w} then
      x[i] ← 0
      GetOptimalChoice2(w, i − 1, C)
    if c_{i-1,w-w_i} + v_i = c_{i,w} then
      x[i] ← 1
      GetOptimalChoice2(w − w_i, i − 1, C)
  else
    Print "Solution:" x[ ]
```

c) Chaque solution (optimale ou non, admissible ou non) est un sous-ensemble de

$$\{1, 2, \ldots, n\}.$$

Donc, l'affirmation est vraie parce que $|\text{Pot}(\{1, 2, \ldots, n\})| = 2^n$.

d) On peut choisir $v_i = 1$ et $w_i = 1$ pour $i = 1, \ldots, n$ et $W = n/2$. Les solutions optimales sont alors des sous-ensembles de $\{1, \ldots, n\}$ de taille $n/2$, et il y en a $\binom{n}{n/2}$. Il est facile de vrifier que

$$\binom{n}{n/2} > 2^{n/2},$$

d'où le résultat.

4. La suite de Fibonacci

a) Il y a la boucle principale qui est parcourue $n - 3$, donc $O(n)$ fois, mais il faut être attentif au fait qu'une itération ne se fait pas á coût constant, parce que les nombres qui apparaissent dans les calculs croissent rapidement. Nous ne considérons ici que le coût de l'addition.[1] Dans la i-ème itération, l'opération

$$a_3 \leftarrow a_2 + a_1$$

calcule F_{i+2}, c'est donc une addition à

$$\log(F_{i+2}) = \log(\varphi^{i+2}) = (i + 2) \cdot \log(\varphi) = O(i)$$

bits, et se fait donc en $O(i)$. Comme $i \leq n$, nous pouvons dire grossiérement qu'une seule itération coûte donc $O(n)$, ce qui implique que l'algorithme est $O(n^2)$.

[1] Remarquons que les opérations restantes, les affectations comme p.ex. $a_1 \leftarrow a_1$, se font en temps $O(N)$ (où N est la taille des nombres) si une implémentation triviale est utilisée, et ne changeront pas l'ordre des calculs en question. En pratique, la copie de nombres se fait souvent avec une stratégie paresseuse (delayed copying), qui, dans notre cas particulier, résulte en un coût constant pour ces opérations.

Remarque: Rappelons que la formule de sommation de Gauß vue au chapitre 0 du cours implique que $\sum_{i=1}^{n} i = \Omega(n^2)$. Pour cette raison, notre estimation grossière du temps de parcours de l'algorithme est en verité précise (dans la mesure où elle ne peut pas être améliorée en l'ordre).

b) Il est clair que c'est vrai pour $n = 0$. Par induction, pour $n > 0$, nous avons

$$\vec{a}_n = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \vec{a}_{n-1} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n + F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix}.$$

La formule fermée est

$$\vec{a}_n = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

La preuve par induction est triviale, et nous l'omettons.

c) Nous commençons par rappeler l'algorithme pour élever des nombres à des puissances quelconques (pour des détails, voir l'exercice 2 de la série 1). Soit $a \in \mathbb{Z}$ et $n \in \mathbb{N}$. Pour calculer a^n, remarquons d'abord que

$$a^2 = a \cdot a \qquad a^4 = a^2 \cdot a^2 \qquad a^8 = a^4 \cdot a^4 \qquad \cdots \qquad a^{2^i} = a^{2^{i-1}} \cdot a^{2^{i-1}}.$$

Il s'ensuit que nous avons besoin de $\lfloor \log(n) \rfloor$ multiplications pour calculer

$$a, a^2, a^4, \ldots, a^{2^{\lfloor \log(n) \rfloor}}.$$

Nous pouvons écrire n comme nombre binaire

$$n = i_{\lfloor \log(n) \rfloor} 2^{\lfloor \log(n) \rfloor} + i_{\lfloor \log(n) \rfloor - 1} 2^{\lfloor \log(n) \rfloor - 1} + \cdots + i_1 2^1 + i_0.$$

Et on voit alors que

$$a^n = a^{\sum_{j=0}^{\lfloor \log(n) \rfloor} i_j 2^j} = \prod_{j=0}^{\lfloor \log(n) \rfloor} a^{i_j 2^j}.$$

Comme le $j^{\text{ème}}$ facteur peut être omis si $i_j = 0$, ce produit est formé d'au plus $\lfloor \log(n) \rfloor$ facteurs. En tout, nous avons donc besoin de $2 \lfloor \log(n) \rfloor$ multiplications pour trouver a^n.

Remarquons maintenant que la méthode comme esquissée ci-dessus ne dépend pas du fait que a est un entier: nous pouvons aussi bien remplacer a par une matrice, par exemple la matrice

$$A := \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

et ensuite utiliser la même technique pour calculer A^n en $O(\log(n))$ produits de matrices 2×2.

Par les points précédents, pour trouver F_n, il suffit de calculer A^{n-1}. Pour ce faire, nous devons effectuer $O(\log(n))$ produits matriciels. Comme dans la partie a) de l'exercice, le coût d'une multiplication n'est pas constant, parce que les valeurs de la matrice croissent: on sait que les composantes de A^n sont de longueur $O(\log(n))$, parce que le fait que les composantes de A^n ne peuvent être que des entiers non-négatifs combiné avec la formule établie au point (1b) permet de borner les composants sont tous supérieurement par $\leq F_{n+2}$.

En résumé:

- Les multiplications matricielles opèrent sur des matrices 2×2 de coefficients de longueur $O(n)$. Une telle multiplication coûte donc $O(n \log n)$.
- Nous avons besoin de $O(\log(n))$ multiplications pour calculer F_n.
- Le temps de calcul est donc $O(n \log^2(n))$.

Remarquons que cette estimation montre déjà que cet algorithme est bien meilleur que l'algorithme du point a).

La réponse "$O(n \log^2(n))$" est celle que nous attendions à cet exercice; elle donne déjà un bon sentiment du comportement de cet algorithme.

Néanmoins, on peut donner une estimation plus fine du temps de parcours de cet algorithme, la raison étant que les tailles des nombres croissent exponentiellement vite avec le nombre d'itérations (contrairement à l'algorithme du point 1a):

- La matrice A a des coefficients de longueur ≤ 1
- La matrice A^2 a des coefficients de longueur ≤ 2
- La matrice A^4 a des coefficients de longueur ≤ 4
- $\cdots$
- La matrice $A^{\lfloor \log(n) \rfloor}$ a des coefficients de longueur $\leq 2^{\lfloor \log(n) \rfloor}$.

Le coût des i premières itérations est donc

$$\leq c \cdot \sum_{j=1}^{i} 2^j \log(2^j) = c \cdot \sum_{j=1}^{i} 2^j \cdot j,$$

où c est une constante (qui ne nous intéressera pas). Les $O(\log(n))$ opérations de l'algorithme nous coûtent donc au plus

$$c \sum_{j=0}^{\lfloor \log(n) \rfloor} 2^j \cdot j \leq c \sum_{j=1}^{\lfloor \log(n) \rfloor} 2^j \log(n) = c \log(n) \sum_{j=1}^{\lfloor \log(n) \rfloor} 2^j = c \log(n)(2^{\lfloor \log(n) \rfloor + 1} - 1) \leq c \log(n) \cdot 2 \cdot n.$$

Donc, c'est un algorithme $O(n \log(n))$.

d) Il est raisonnable de supposer que la machine a besoin de temps $\Omega(n)$ pour écrire n bits d'output ("écrire un chiffre dure un certain temps"). Il s'ensuit que nous avons besoin d'un temps $\Omega(n)$ seulement pour donner l'output F_n (comme F_n a $\theta(n)$ chiffres), en ne considérant aucun calcul effectué.

Tout algorithme pour ce faire est donc $\Omega(n)$, et en particulier il n'y en a aucun en $o(n)$.