

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

Corrigé de la série 3

5 Octobre 2007

1. Running time

Le but de cet exercice est d'analyser les temps de parcours d'algorithmes. Dans ces exemples la valeur de la variable a est égale au nombre d'opérations effectuées (sauf pour f_4), elle permet donc de suivre le temps de parcours. De manière générale :

- Il est clair qu'une boucle de longueur n

```

1: for  $i = 1, \dots, n$  do
2:    $a \leftarrow a + 1$ 

```

va incrémenter la variable a de n .

Quand on a plusieurs boucles :

- Si elles sont l'une après l'autre on fait une somme :

```

1: for  $i = 1, \dots, n$  do
2:    $a \leftarrow a + 1$ 
3: for  $i = 1, \dots, m$  do
4:    $a \leftarrow a + 1$ 

```

va incrémenter a de $n + m$.

- Si elles sont l'une dans l'autre on fait une multiplication :

```

1: for  $i = 1, \dots, n$  do
2:   for  $i = 1, \dots, m$  do
3:      $a \leftarrow a + 1$ 

```

va incrémenter a de $n \cdot m$.

- a) • Dans f_1 on voit que pour chaque i , a est incrémenté i fois. Cette opération est répétée pour chaque $i = 1, \dots, n$, on a donc au total

$$f_1(n) = 1 + 2 + 3 + \dots + n = \frac{n \cdot (n + 1)}{2}.$$

- Dans f_2 , pour chaque i , a est incrémenté $2n$ fois, puis $\lfloor \sqrt{n} \rfloor$ fois, donc au total $2n + \lfloor \sqrt{n} \rfloor$ fois. Puisque cette opération est répétée pour chaque i (donc n fois), on a

$$f_2(n) = n \cdot (2n + \lfloor \sqrt{n} \rfloor).$$

- $f_3(2) = 5$. $f_3(3) = 14$.

On observe d'abord que puisque la variable k n'est pas utilisée à l'intérieur des boucles, la ligne 4 est équivalente à

for $k = 1, \dots, i$ **do**

(on répète l'opération à l'intérieur de la boucle i fois, c'est égal si k va de $j + 1$ à $j + i$ où de 1 à i).

Ainsi, pour chaque i on incrémente a i^2 fois. Cette opération est répétée pour chaque $i = 1, \dots, n$, donc on a au total

$$\begin{aligned} f_3(n) &= 1^2 + 2^2 + 3^2 + \dots + n^2 \\ &= \frac{n \cdot (n+1) \cdot (2n+1)}{6}. \end{aligned}$$

En effet nous avons montré dans la série 1 (par induction, exercice 4.b) qu'on avait $\sum_{i=1}^n i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$.

On voit que cette formule est en corrélation avec les résultats $f_3(2) = 5$ et $f_3(3) = 14$.

• Nous voyons d'abord que la ligne 3 sera exécutée n fois, et que chaque exécution incrémente a de $\ln(n)$. Donc au début de la ligne 4, la variable a vaudra $n \cdot \ln(n)$.

Ensuite, pour chaque i la ligne 6 sera exécutée $n - i + 1$ fois. Par exemple si $n = 5$ et $i = 2$, elle sera exécutée pour $j = 2, 3, 4, 5$ donc 4 fois ce qui est bien égal à $n - i + 1$. Donc la ligne 6 sera exécutée un total de

$$\sum_{i=1}^n (n - i + 1) = n + (n - 1) + (n - 2) + \dots + 2 + 1 = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

fois. Puisqu'à chaque exécution de la ligne 6 a est incrémentée n fois, toutes ces exécutions ensemble incrémentent a de $\frac{n^2(n+1)}{2}$.

En combinant ce résultat avec l'effet de la ligne 3, nous obtenons :

$$f_4(n) = n \cdot \ln(n) + \frac{n^2(n+1)}{2}.$$

- b) • f_1 est $\theta(n^2)$
 • f_2 est $\theta(n^2)$
 • f_3 est $\theta(n^3)$
 • f_4 est $\theta(n^3)$

2. Algorithmme

- a) n est une puissance de 2. Supposons qu'on a $n = 2^k$.
- 1. On calcule $a * a = a^2$, qu'on stocke en mémoire. Notons que $2 = 2^1$ et qu'il s'agit de la 1^{ère} étape.
 - 2. On calcule $a^2 * a^2 = a^4$, qu'on stocke en mémoire. Notons que $4 = 2^2$ et qu'il s'agit de la 2^{ème} étape.

- 3. On calcule $a^4 * a^4 = a^8$, qu'on stocke en mémoire. Notons que $8 = 2^3$ et qu'il s'agit de la 3^{ème} étape.

\vdots \vdots

- k. On calcule $a^{2^{k-1}} * a^{2^{k-1}} = a^{2^k} = a^n$, et on a donc trouvé a^n . $n = 2^k$, c'est donc bien la k^{ème} étape.

Si $n = 2^k$, cette méthode permet de trouver a^n en $k = \log_2(n)$ opérations.

b) Soit $n \in \mathbb{N}$. On pose $k = \lfloor \log_2(n) \rfloor$. Puisque $k \leq \log_2(n) < k + 1$, on a

$$2^k \leq n < 2^{k+1}$$

Nous savons de la partie a) que nous pouvons calculer assez rapidement a^ℓ quand ℓ est une puissance de 2. Tout nombre n peut s'exprimer comme une somme de puissances de 2 (c'est sa *représentation binaire*). Si $k = \lfloor \log_2(n) \rfloor$, alors n s'exprime sous la forme

$$n = b_k \cdot 2^k + b_{k-1} \cdot 2^{k-1} + \dots + b_1 \cdot 2 + b_0, \tag{1}$$

où $b_i \in \{0, 1\}$ pour tout i .

Nous avons par exemple

$$27 = 16 + 8 + 2 + 1 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0,$$

27 s'écrit donc 11011 en base 2 ($b_4 = 1, b_3 = 1, b_2 = 0, b_1 = 1, b_0 = 1$).

L'idée est procéder en 2 étapes :

- **Etape 1** : Nous calculons les valeurs $a^2, a^{2^2}, \dots, a^{2^{k-1}}, a^{2^k}$. Il faut à cette étape k opérations (nous utilisons l'algorithme de la partie a) pour calculer a^{2^k} , et voyons qu'il calcule aussi au passage les autres valeurs cherchées).

- **Etape 2** : Nous utilisons ces valeurs pour calculer a^n à partir de sa représentation binaire. Puisque

$$n = b_k \cdot 2^k + \dots + b_0,$$

nous avons

$$a^n = a^{b_k \cdot 2^k} * \dots * a^{b_0}. \tag{2}$$

Puisqu'il y a au plus $k + 1$ termes (de b_0 à b_k) dans (2), il faudra faire au plus k multiplications (il peut y en avoir moins, puisque si $b_i = 0$ alors $a^{b_i \cdot 2^i} = 1$ la multiplication n'est donc pas nécessaire). Plus précisément, si s est le nombre de 1 dans la représentation binaire de n alors il faudra $s - 1$ multiplications.

Ainsi le nombre total d'opération pour les deux étapes est au plus

$$k + k = \lfloor \log_2(n) \rfloor + \lfloor \log_2(n) \rfloor = 2 \cdot \lfloor \log_2(n) \rfloor.$$

Exemple 1 : Prenons $n = 27$, donc $k = \lfloor \log_2(27) \rfloor = 4$.

Etape 1 : Nous calculons a^2, a^4, a^8 et a^{16} :

$$\begin{aligned} a * a &= a^2 && (1 \text{ opération}) \\ a^2 * a^2 &= a^4 && (1 \text{ opération}) \\ a^4 * a^4 &= a^8 && (1 \text{ opération}) \\ a^8 * a^8 &= a^{16} && (1 \text{ opération}) \end{aligned} \tag{3}$$

Il nous a fallu 4 opérations pour cette première étape.

Etape 2 : Nous utilisons la représentation binaire de 27 : Puisque

$$27 = 16 + 8 + 2 + 1,$$

nous déduisons que

$$a^{27} = a^{16} * a^8 * a^2 * a^1,$$

il nous a donc fallu 3 opérations pour cette deuxième étape.

Ainsi Le nombre total d'opération pour les deux étapes est $4 + 3 = 7$, ce qui est bien plus petit que $2 \cdot \lfloor \log_2(n) \rfloor = 2k = 8$.

Exemple 2 : Prenons $n = 63$, donc $k = \lfloor \log_2(63) \rfloor = 5$.

Etape 1 : Nous calculons a^2, a^4, a^8, a^{16} et a^{32} , il nous faut 5 opérations (voir (3)).

Etape 2 : Nous utilisons la représentation binaire de 63. Puisque

$$63 = 32 + 16 + 8 + 4 + 2 + 1,$$

sa représentation binaire est 111111, et

$$a^{63} = a^{32} * a^{16} * a^8 * a^4 * a^2 * a^1.$$

il nous a donc fallu 5 opérations pour cette deuxième étape.

Ainsi le nombre total d'opération pour les deux étapes est $5 + 5 = 10$, et nous avons bien $2 \cdot \lfloor \log_2(n) \rfloor = 2k = 10$.

Remarquons que le "pire des cas" pour cet algorithme survient lorsque la représentation binaire de n ne contient que des 1 (comme pour 63), c'est à dire lorsque n peut s'exprimer sous la forme $2^\ell - 1$.

Nous concluons en remarquant que cet algorithme n'est pas le meilleur possible pour tous les n . Par exemple, pour $n = 15$ il lui faut 6 opération, alors que nous avons vu dans la partie c) qu'on pouvait calculer a^{15} en 5 opérations.

3. L'algorithme de Karatsuba

- a) Il faut multiplier chaque coefficient de f par chaque coefficient de g , donc $4 \cdot 4 = 16$ multiplications.
- b) On a $f(x) = f_0(x) + f_1(x) * x^2$, et de même pour $g(x)$, ce qui nous donne :

$$\begin{aligned} f_0 &= 1 + 2x \\ f_1 &= 1 + 2x \\ g_0 &= 2 + 2x \\ g_1 &= 1 + x \\ u &= 2 + 4x \\ v &= 3 + 3x \end{aligned}$$

c) On veut trouver

$$f(x) \cdot g(x) = h(x) = h_0(x) + h_1(x) \cdot x^2 + h_2(x) \cdot x^4$$

où $h_0(x)$, $h_1(x)$ et $h_2(x)$ sont des polynômes de degré 2 (voir le cours).

On calcule d'abord $h_0 = f_0 \cdot g_0 = (1 + 2x)(2 + 2x)$. Pour ce, on utilise de nouveau l'algorithme de Karatsuba. Posons $h_0 = a + bx + cx^2$. On fait $a = 1 \cdot 2 = 2$, $b = 2 \cdot 2 = 4$ et $A = (1 + 2) \cdot (2 + 2) = 12$. Finalement, pour obtenir b on fait $b = A - a - c = 12 - 2 - 4 = 6$. On a donc trouvé $h_0 = 2 + 6x + 4x^2$, en 3 multiplications.

Ensuite il nous faut trouver $h_2 = f_1 \cdot g_1 = (1 + 2x)(1 + x)$. On utilise de nouveau l'algorithme de Karatsuba, pour obtenir $h_2 = (1 + 3x + 2x^2)$, en 3 multiplications.

Nous devons ensuite calculer $A = (f_0 + f_1) \cdot (g_0 + g_1) = (2 + 4x) \cdot (3 + 3x)$. On peut de nouveau utiliser l'algorithme de Karatsuba, pour obtenir $A = 6 + 18x + 12x^2$ en 3 multiplications.

On obtient $h_1 = A - h_0 - h_2 = (6 + 18x + 12x^2) - (2 + 6x + 4x^2) - (1 + 3x + 2x^2) = 3 + 9x + 6x^2$.

Ainsi, on a trouvé :

$$\begin{aligned} h(x) &= h_0(x) + h_1(x) \cdot x^2 + h_2(x) \cdot x^4 \\ &= (2 + 6x + 4x^2) + (3 + 9x + 6x^2) \cdot x^2 + (1 + 3x + 2x^2) \cdot x^4 \\ &= 2 + 6x + 7x^2 + 9x^3 + 7x^4 + 3x^5 + 2x^6 \end{aligned}$$

On remarque que pour trouver h à partir de h_0 , h_1 , et h_2 nous n'avons pas eu besoin de faire de multiplications (seulement des additions).

Ils nous a donc fallu exactement 9 multiplications de réels (remarquons cependant que nous avons aussi dû faire des additions de réels).

4. Elever une matrice au carré

a) Nous connaissons a, c, b et d et voulons calculer :

$$A^2 = \begin{pmatrix} a^2 + bc & ab + bd \\ ca + dc & cb + d^2 \end{pmatrix}$$

Ecrit de cette façon il nous faudrait 8 multiplications pour calculer A^2 . Elles ne sont cependant pas toutes nécessaires, en effet on peut réécrire notre matrice comme suit :

$$A^2 = \begin{pmatrix} a^2 + bc & b(a + d) \\ c(a + d) & bc + d^2 \end{pmatrix}$$

La multiplication sur \mathbb{R} est commutative et distributive, cette dernière matrice est donc bien égale à A^2 . On voit que bc apparaît deux fois, mais il nous suffit de le calculer une seule fois. Les factorisations nous permettent également de remplacer deux multiplications par une seule. Pour calculer a^2 nous avons donc besoin des 5 multiplications suivantes : $a * a, d * d, b * c, b * (a + d), c * (a + d)$.

On remarque qu'il nous faut aussi 4 additions d'éléments de \mathbb{R} .

b) Malheureusement cet algorithme ne peut pas être généralisé pour élever des matrices de taille quelconque au carré. Nous illustrerons maintenant pourquoi, dans ce cas, on ne peut pas appliquer la technique diviser-pour-régner pour généraliser l'algorithme.

De manière générale, un algorithme diviser-pour-régner réduit les problèmes de taille n à des sous-problèmes du même type de plus petite taille. Par exemple, l'algorithme de Karatsuba réduit le calcul de deux polynômes de degré $< 2n$ au calcul de trois produits de polynômes de degré $< n$. De manière similaire, l'algorithme de Strassen permet de réduire le calcul du produit matriciel de deux matrices de taille $2n \times 2n$ à des produits matriciels de taille n .

Supposons que a, b, c et d sont elles-mêmes des matrices $n \times n$, et que la matrice A que nous aimerions élever au carré est donc une matrice $2n \times 2n$.

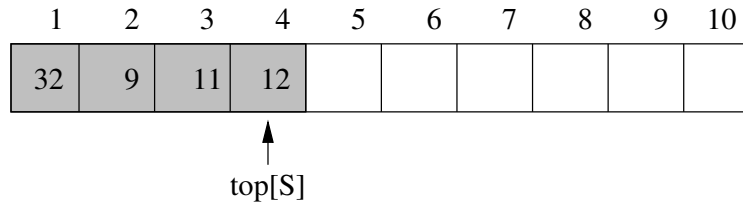
La technique diviser-pour-régner consisterait maintenant en la réduction du calcul du carré de la $2n \times 2n$ -matrice A au calcul de carrés de matrices de taille $n \times n$.

Mais les produits des a, b, c, d comme obtenus sous le point précédent ne sont pas tous des carrés ! Nous ne pourrions donc pas récursivement utiliser le même algorithme pour effectuer ces calculs. Nous aurons besoin d'un algorithme de multiplication de matrices général à la place.

Un deuxième problème est que nous avons utilisé la commutativité de la multiplication dans \mathbb{R} pour trouver nos produits. Les formules ne resteront donc pas valides si a, b, c et d sont des matrices, parce que le produit matriciel n'est pas, en général, commutatif.

5. Stacks

a) Après ces opérations on a :



b) On voit qu'il y a une parenthèse fermante de trop, l'output attendu est donc "incorrect".

L'algorithme du cours ferait les opérations suivantes :

- (1) Push([)
- (2) Push(()
- (3) Pop() l'élément enlevé est "(" , il correspond bien à ")" donc nous continuons
- (4) Push(()
- (5) Push(()
- (6) Pop() l'élément enlevé est "(" , il correspond bien à ")" donc nous continuons
- (7) Pop() l'élément enlevé est "(" , il correspond bien à ")" donc nous continuons
- (8) Pop() l'élément enlevé est "[" (le seul qui reste sur le stack), il ne correspond pas à ")" l'algorithme retourne donc "incorrect"