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

Corrigé de la série 3 5 Octobre 2009

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, ..., n do
2:     a ← 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, ..., n do
2:     a ← a + 1
3: for i = 1, ..., m do
4:     a ← a + 1
```

va incrémenter a de $n + m$.

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

```
1: for i = 1, ..., n do
2:     for i = 1, ..., m do
3:         a ← 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 $i = 1, \ldots, n$, on a donc au total

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

• 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

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

• $f_3(2) = 5$. $f_3(3) = 14$.

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

$$\textbf{for } k = 1, \ldots, i \textbf{ do}$$

(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 $i = 1, \ldots, n$, donc on a au total

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

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 correlation avec les résultats $f_3(2) = 5$ et $f_3(3) = 14$.

• Nous voyons d'abord que la ligne 3 sera executée n fois, et que chaque execution 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 executée $n - i + 1$ fois. Par exemple si $n = 5$ et $i = 2$, elle sera executée pour $j = 2, 3, 4, 5$ donc 4 fois ce qui est bien égal à $n - i + 1$. Donc la ligne 6 sera executée un total de

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

fois. Puisqu'à chaque execution de la ligne 6 a est incrémenté n fois, toutes ces executions ensemble incrémenteront 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. 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 necessaires, en effet en 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 apparait deux fois, mais il nous suffit 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 polynomes de degré $< 2n$ au calcul de trois produits de polynômes de degre $< n$. 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 pourrons 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.

3. Algorithme

a) On effectue les 2 opérations suivantes:

- *1.* On calcule $a * a = a^2$ qu'on stocke en mémoire. On a donc a et a^2 en mémoire.
- *2.* On calcule $a^2 * a^2 = a^4$. On a donc a^4 en mémoire.

b) On effectue les opérations suivantes:

- *1.* On calcule $a * a = a^2$, qu'on stocke en mémoire.
- *2.* On calcule $a * a^2 = a^3$, qu'on stocke en mémoire.
- *3.* On calcule $a^2 * a^2 = a^4$, qu'on stocke en mémoire.
- *4.* On calcule $a^4 * a^3 = a^7$, et on a donc trouvé a^7.

c) On effectue les opérations suivantes:

- *1.* On calcule $a * a = a^2$, qu'on stocke en mémoire.
- *2.* On calcule $a * a^2 = a^3$, qu'on stocke en mémoire.
- *3.* On calcule $a^3 * a^2 = a^5$, qu'on stocke en mémoire.
- *4.* On calcule $a^5 * a^5 = a^{10}$, qu'on stocke en mémoire.
- *5.* On calcule $a^5 * a^{10} = a^{15}$, et on a donc trouvé a^{15}.

d) 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 $1^{\text{ère}}$ 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 $2^{\text{ème}}$ 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 $3^{\text{ème}}$ étape.

$$\vdots \qquad \vdots \qquad \vdots$$

- *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 $k^{\text{ème}}$ étape.

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

e) 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 d) 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

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

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

L'idée est proceder en 2 étapes:

• **Etape 1:** Nous calculons les valeurs $a^2, a^{2^2}, \ldots, a^{2^{k-1}}, a^{2^k}$. Il faut à cette étape k opérations (nous utilisons l'algorithme de la partie d) 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

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

nous avons

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

Puisqu'il y a au plus $k+1$ termes (de b_0 à b_k) dans (**??**), 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 necessaire). Plus précisément, si s est le nombre de 1 dans la représenation 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}:

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

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 (**??**)).

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

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

sa repreśentation 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.

4. 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.

5. Permutations avec stacks et files d'attente

a)
- $(1, 2, 3, 5, 4)$ est possible par la suite suivante d'instructions:
  `Push, Pop, Push, Pop, Push, Pop, Push, Push, Pop, Pop.`
- $(2, 3, 5, 4, 1)$ est possible par:
  `Push, Push, Pop, Push, Pop, Push, Push, Pop, Pop, Pop.`
- Mais il n'y a aucune suite d'instructions qui sort $(3, 1, 2, 5, 4)$.

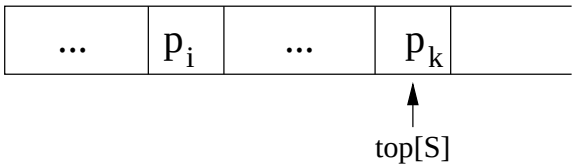
b) Nous devons d'abord fixer l'algorithme qui résout le problème et montrer ensuite qu'il fonctionne bien si la permutation vérifie la propriété exigée. Nous fixons la permutation $(p_1, \ldots, p_n)$ et considérons l'algorithme suivant:

1: $j \leftarrow 1$
2: $i \leftarrow 1$
3: **for** $i = 1, \ldots, n$ **do**
4: **while** $j \leq p_i$ **do**
5: `Push`(S, j)
6: $j \leftarrow j + 1$
7: `Pop`(S)

L'ordre des push effectué de cet algorithme est bien `Push`$(S, 1)$, `Push`$(S, 2), \ldots,$ `Push`(S, n) parce que j est incrémenté après chaque opération `Push`. Aussi le "while" de l'étape 4 se termine forcément parce que j est incrémenté en chaque itération et testé contre une valeur p_i fixe.

Le seul endroit possible d'erreur de l'algorithme est alors l'instruction `Pop` à l'étape 8. Montrons d'abord qu'il n'est pas possible que `Pop` résulte en un stack underflow: Supposons que nous nous trouvons à l'itération i juste avant le `Pop`. Alors il existe ℓ, $1 \leq \ell \leq i$, tel que $p_\ell \geq i$. Si ce n'était pas le cas, on avait que $\{p_1, \ldots p_i\} \subseteq \{1, \ldots, i-1\}$. Comme les p_k sont distincts, l'ensemble $\{p_1, \ldots, p_i\}$ est de taille i et ne peut donc pas être inclus dans $\{1, \ldots, i-1\}$ qui est de taille $i - 1$.

La dernière possibilité d'erreur est finalement donc que `Pop` ne rend pas la valeur p_i attendue. Nous supposerons par la suite que l'itération i est la première itération d'échec de l'algorithme. Comme le "while" nous assure que p_i doit se trouver sur le stack, ceci n'arrive que s'il y a une autre valeur encore au dessus du stack, i.e. qu'on se trouve dans la situation suivante:



Nous avons $i < k$ puisque p_k se trouve encore sur le stack. D'autre part, comme les éléments sur le stack sont stockés de manière croissant, nous avons $p_i < p_k$. Une telle situation sur le stack ne peut arriver que si à une itération précédente $l < i$, la valeur p_l devait être sorti avec $p_l > p_k$. En résumé:

$$l < i < k,$$
$$p_i < p_k < p_l,$$

ce qui correspond à l'énoncé.

c) Pour n'importe quelle suite d'instructions, l'output sera toujours $(1, 2, 3, 4, 5)$. Parce que dans une file d'attente l'élément enlevé est l'élément le plus vieux. Alors, 1 est toujours le premier élément qui sort la file, 2 est toujours le deuxième élément qui sort la file, etc.