ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

Corrigé de la série 3 11 Octobre 2010

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

for $k = 1, \ldots, i$ **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 $i = 1, \ldots, n$, 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 pouvons montré par induction 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. Relations de récurrence

a) On utilise le Théorème 2.1 du cours. On voit que la condition (a) est vérifiée puisque $T(n)$ est montone croissante, et la condition (b) est vérifiée avec

$$\begin{array}{rcl} a & = & 10 \\ b & = & 1/2 \\ c & = & 3 \\ d & = & 10^3 \end{array}$$

On voit ensuite que $a^b = \sqrt{10} > 3 = c$, on est donc dans le cas (3) du Théorème, donc $T(n) = O(n^b)$, i.e. $T(n) = O(\sqrt{n})$

b) Quand $n = 2^k$ on a:

$$\begin{array}{rcl} T(2^k) & = & T(2 \cdot 2^{k-1}) \\ & = & 2^{k-1} \cdot T(2^{k-1}) \\ & = & 2^{k-1} \cdot T(2 \cdot 2^{k-2}) \\ & = & 2^{k-1} \cdot 2^{k-2} \cdot T(2^{k-2}) \\ & \vdots & \vdots \\ & = & \prod_{i=1}^{k-1} 2^i \cdot T(1) \\ & = & 2^{\sum_{i=1}^{k-1} i} \cdot T(1) \\ & = & 2^{\frac{k(k-1)}{2}} \cdot T(1) \\ & = & 2^{\frac{k(k-1)}{2}} \end{array}$$

Ensuite si on pose $k = \lfloor \log_2 n \rfloor$, on a

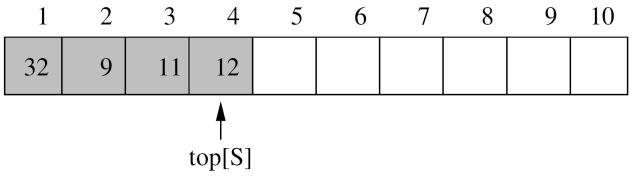
$$\begin{array}{lcll} n \leq 2^{k+1} & \Longrightarrow & T(n) \leq T(2^{k+1}) & \\ & \Longrightarrow & T(n) \leq 2^{\frac{k(k+1)}{2}} & \\ & \Longrightarrow & T(n) \leq 2^{\frac{\log_2 n(\log_2(n)+1)}{2}} & \text{(puisque } k \leq \log_2 n\text{)} \end{array}$$

Finalement on voit que

$$\begin{array}{rcl} 2^{\frac{\log_2 n(\log_2(n)+1)}{2}} & = & (2^{\log_2^2 n + \log_2 n})^{1/2} \\ & = & ((2^{\log_2 n})^{log_2 n} \cdot 2^{log_2 n})^{1/2} \\ & = & (n^{\log_2 n} \cdot n)^{1/2} \\ & = & n^{\frac{\log_2(n)+1}{2}} \end{array}$$

3. 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"

4. 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 7. 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.