Complexité
Corrigé

Fabrice Rossi

12 mars 2012

1 Correction de l'exercice 1.1

On considère donc le programme suivant :

```
1  i = 0
2  j = 0
3  while(i < n) {
4    if( i % 2 == 0) {
5      j = j + 1
6    } else {
7      j = j / 2
8    }
9    i = i + 1
10 }
```

Le programme est constitué d'une phase d'initialisation (lignes 1 et 2), puis d'une boucle. On détermine d'abord le nombre d'exécution de la boucle. On remarque que la boucle s'exécute à chaque fois que `i < n`. Or, `i` est initialisé à 0 (ligne 1) et chaque exécution (tour) de la boucle incrémente `i` de 1 (ligne 9). La boucle s'exécute donc pour les valeurs de `i` 0, 1, etc. jusqu'à `n-1`, soit donc `n` fois.

On constate ensuite que le corps de la boucle (lignes 4 à 9), contient un nombre d'instruction qui ne dépend ni de la valeur de `i` ni de celle de `j`. En effet, les deux lignes 5 et 7 contiennent chacune deux instructions (un calcul et une affectation). Les autres lignes (4 et 9) sont toujours exécutées. De ce fait, le temps d'exécution du corps de la boucle est donc un $\Theta(1)$.

Comme la boucle s'exécute `n` fois, le temps d'exécution du programme est alors en $\Theta(n)$.

2 Correction de l'exercice 1.2

Le programme étudié est le suivant :

```
1  x = 0
2  for(int i = 1 ; i < tab.length-1 ; i++) {
3    for(int j = 0 ; j < 3 ; j++) {
4       x = x + tab[i - 1 + j] * (j+1)
5    }
6  }
```

L'analyse commence comme toujours par la boucle la plus interne, ici celle de lignes 3 à 5. La quantité d'opérations réalisé dans un tour de la boucle (ligne 4 essentiellement) ne dépend ni des valeurs des variables `i` et `j`, ni de la taille du tableau `tab` (ou de son contenu). On a en effet :

– une affectation `x=` ;
– trois additions et une soustraction `j+1`, `i-1+j` et `x+` ;
– un accès à une case du tableau `tab` ;
– une multiplication `*(j+1)` ;
– une incrémentation `i++` ;
– une comparaison `j < 3`.

Globalement, un tour de cette boucle réalise donc $\Theta(1)$ opérations.

De plus, la boucle s'exécute exactement trois fois pour `j` prenant les valeurs 0, 1 et 2. Donc le nombre d'opérations réalisées par la boucle complète est un $\Theta(1)$.

On constate ensuite que l'exécution complète de la boucle des lignes 3 à 5 constitue l'essentiel de l'exécution d'un tour de la boucle des lignes 2 à 5 (la boucle externe). Un tour de cette boucle prend donc $\Theta(1)$ opérations. Enfin, cette boucle externe s'exécute pour les valeurs de `i` allant de 1 à `tab.length-2`, bornes incluses. On a donc `tab.length-2` exécutions du corps de la boucle, soit un nombre total d'opérations en (`tab.length-2`) $\times \Theta(1)$, c'est-à-dire Θ(`tab.length`).

3 Correction de l'exercice 1.3

Cet exercice ressemble beaucoup à l'exercice 1.2, avec une différence fondamentale dans la boucle interne. En effet, dans l'exercice 1.2, la boucle interne réalise un nombre constant d'opérations alors que dans le présent exercice, le nombre d'opérations dépend des caractéristiques des données. Le programme est en effet le suivant :

```
1  x = 0
2  for(int i = 0 ; i <= tab.length - filter.length ; i++) {
3    for(int j = 0 ; j < filter.length ; j++) {
4       x = x + tab[i + j] * filter[j]
5    }
6  }
```

Comme dans l'exercice précédent, une exécution de la boucle interne (lignes 3 à 5) réalise $\Theta(1)$ opérations. Mais cette boucle est exécutée `filter.length` fois (pour `j` allant de 0 à `filter.length-1`). Donc le nombre d'opérations de la boucle interne complète est en Θ(`filter.length`).

La boucle externe (lignes 2 à 5) s'exécute elle `tab.length - filter.length+1` fois. Si on note $n =$ `tab.length` et $p =$ `filter.length`, on a donc un coût total en $(n-p) \times \Theta(p)$ et donc en $\Theta((n-p)p)$ (en s'appuyant sur le fait que le corps de la boucle externe consiste essentiellement en l'exécution complète de la boucle interne).

4 Correction de l'exercice 3.1

On étudie la recherche dichotomique dans un tableau trié :

search

```
1  search(x,tab) {
2    if(tab.length == 0) {
3      return -1
4    } else if(tab.length == 1) {
5       if(tab[0] == x) {
6         return 0
7       } else {
8         return -1
9       }
10   } else {
11     pos = tab.length/2
12     if(tab[pos] == x) {
```

```
13        return pos
14      } else if(tab[pos] < x) {
15        subpos = search(x,tab[(pos + 1):(tab.length - 1)])
16        if(subpos >= 0) {
17          return subpos + pos + 1
18        } else {
19          return -1
20        }
21      } else {
22        subpos = search(x,tab[0:(pos - 1)])
23        if(subpos >= 0) {
24          return subpos
25        } else {
26          return -1
27        }
28      }
29  }
```

On note $n = \texttt{tab.length}$ et $T(n)$ le nombre d'opérations réalisé dans le cas le pire par un appel à la fonction `search` avec comme paramètre un tableau de longueur n. La fonction étant récursive, on cherche à établir une relation de récurrence sur $T(n)$.

4.1 Cas $n = 0$

Ce cas est facile à traiter car il correspond simplement aux lignes 2 et 3 du programme. Ces lignes effectuent un nombre constant d'opérations (deux exactement, la comparaison et la transmission du résultat). On a donc

$$T(0) = \Theta(1).$$

4.2 Cas $n = 1$

Ce cas est aussi assez facile à traiter. Il correspond au test de la ligne 2, puis aux lignes 4 à 9. On constate que le nombre d'opérations réalisées est de nouveau constant car chaque cas du test (lignes 6 et 7) conduisent au même travail, à savoir renvoyer une valeur numérique. On a donc de nouveau

$$T(1) = \Theta(1).$$

4.3 Cas $n > 1$

C'est la partie délicate de l'analyse qui correspond aux lignes 11 à 28 (après avoir réalisé les tests en ligne 2 et 4). On fait ici une analyse dans le cas le pire car le nombre d'opérations réalisées dépend fortement de la position de `x` dans le tableau. En particulier, si cette position est `tab.length/2`, on réalise un nombre constant d'opérations seulement. Mais ce n'est bien entendu pas le cas le pire qui correspond au deux autres situations.

On traite d'abord le cas `tab[pos] < x`, soit les lignes 14 à 20. Le programme commence par construire le sous-tableau `tab[(pos + 1):(tab.length - 1)]`. Notons pour l'instant $V(p)$ le nombre d'opérations nécessaires à la construction d'un sous-tableau de longueur p. Dans le cas présent, le sous-tableau est de longueur $(n-1)-(\lfloor n/2 \rfloor+1)+1$, où $\lfloor n/2 \rfloor$ désigne la partie entière de $n/2$. La longueur du sous-tableau est donc de $n - \lfloor n/2 \rfloor - 1$. L'appel à `search` engendre ainsi un nombre d'opérations de $T(n - \lfloor n/2 \rfloor - 1)$. Après l'exécution de l'appel récursif, le programme passe aux lignes 16 à 20, ce qui engendre $\theta(1)$ opérations dans le cas le pire (on est bien dans une analyse dans le cas le pire puisqu'on a une ligne 17 qui engendre plus d'opérations que la ligne 19 du cas contraire). Finalement, le nombre d'opérations pour le cas `tab[pos] < x` est donc de

$$T(n - \lfloor n/2 \rfloor - 1) + V(n - \lfloor n/2 \rfloor - 1) + \Theta(1).$$

Le cas `tab[pos] > x` se traite de la même façon, mais en considérant le tableau `tab[0:(pos - 1)]` de taille $\lfloor n/2 \rfloor$, ce qui conduit à un nombre d'opérations de

$$T(\lfloor n/2 \rfloor) + V(\lfloor n/2 \rfloor) + \Theta(1).$$

On peut affiner ces calculs en tenant compte de la parité de n :

$n = 2k$: on a alors $\lfloor n/2 \rfloor = k$ et donc dans le premier cas

$$T(k-1) + V(k-1) + \Theta(1)$$

opérations, et dans le deuxième cas

$$T(k) + V(k) + \Theta(1)$$

opérations. Si on fait l'hypothèse naturelle que T et V sont des fonctions croissantes, l'analyse dans le cas le pire conduit à la formule de récurrence suivante :

$$T(2k) = T(k) + V(k) + \Theta(1).$$

$n = 2k + 1$: on a alors $\lfloor n/2 \rfloor = k$ et donc dans les deux cas

$$T(k) + V(k) + \Theta(1)$$

opérations, soit une relation de récurrence en

$$T(2k+1) = T(k) + V(k) + \Theta(1).$$

Donc finalement, on obtient la récurrence suivante :

$$T(n) = T(\lfloor n/2 \rfloor) + V(\lfloor n/2 \rfloor) + \Theta(1).$$

4.4 Application du théorème maître

On termine l'exercice en appliquant le théorème maître, en tenant compte des hypothèses sur V :

1. Si on suppose que l'extraction d'un sous-tableau se fait en temps constant, on a $V(p) = \Theta(1)$ pour tout p. La récurrence devient alors

   $$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1).$$

   On est dans la situation du théorème maître avec $a = 1$, $b = 2$ et $f(n) = \Theta(1)$. Or, $n^{\log_b a} = 1$, ce qui veut dire que $f(n) = \Theta(n^{\log_b a})$ et donc que nous sommes dans le cas 2 du théorème. On a alors $T(n) = \Theta(n^{\log_b a} \log n)$, soit

   $$T(n) = \Theta(\log n).$$

2. Si on suppose que l'extraction d'un sous-tableau demande un temps proportionnel à la longueur du sous-tableau, on a $V(p) = \Theta(p)$. Donc la récurrence devient

   $$T(n) = T(\lfloor n/2 \rfloor) + \Theta(\lfloor n/2 \rfloor).$$

   Comme dans le cas précédent, on applique le théorème maître à $a = 1$ et $b = 2$. Or $\Theta(\lfloor n/2 \rfloor) = \Theta(n) = \Theta(n^{\log_b a + 1})$. De plus, on a naturellement $V(\lfloor n/2 \rfloor) \simeq \frac{1}{2} V(n)$, ce qui nous place dans le cas 3 du théorème maître. On a donc

   $$T(n) = \Theta(n).$$

On constate donc que la recherche dichotomique n'est intéressante que si on peut extraire un sous-tableau en temps constant. Ceci se fait sans difficulté dans la plupart des langages de programmation, soit en utilisant des pointeurs (en C par exemple), soit plus en transmettant des bornes de recherche dans l'appel récursif (en Java par exemple).