IFT2015 hiver 2010 — Notes de cours

Miklós Csűrös

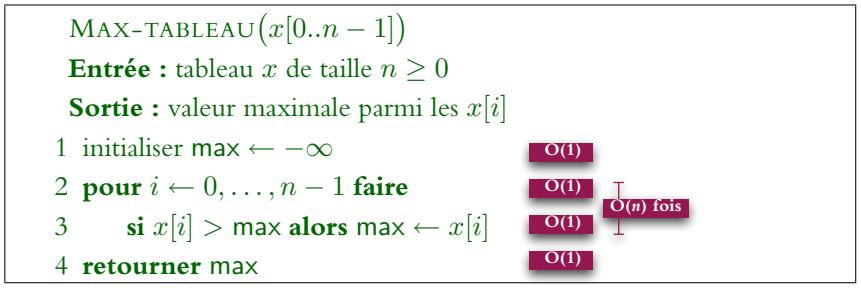
25 janvier 2010

3 Analyse d'algorithmes

3.1 Temps de calcul

Pour caractériser une structure de données, on analysera souvent le temps de calcul pour les **opérations** différentes. Typiquement, on veut arriver à un résultat tel que «l'opération UNION prend $O(\log n)$ temps au pire cas» où n mesure la taille de la strucure. Ce qu'on veut dire par la «taille», dépend du contexte. Ici, n est le nombre d'éléments dans une structure Union-Find. De même façon, on caractérisera un **algorithme** en disant que «algorithme Tel-et-tel prend $O(n)$ temps au pire cas», où n mesure la taille de l'entrée. En général, on veut arriver à une borne asymptotique en fonction de la taille de l'entrée et celle de la sortie.

Exemple 3.1. On prend l'exemple de rechercher le maximum dans un tableau.



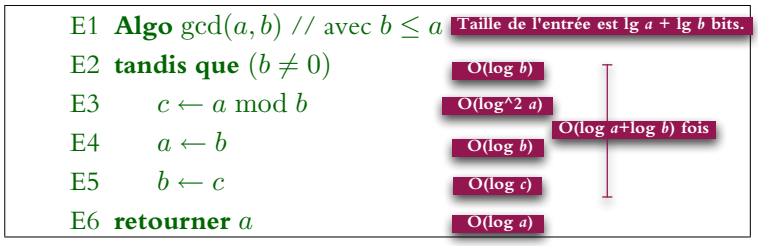
Le temps de calcul pour un tableau de longueur n (=taille de l'entrée) est $T(n) = O(1)+\big(O(1)+O(1)\big)\cdot O(n)+O(1)$, donc $T(n) = O(n)$ (v. règles de l'arithmétique avec O). L'algorithme prend un temps linéaire dans la taille de l'entrée (dans tous les cas). □

Exercice 3.1. *Écrire le pseudocode pour un algorithme qui calcule la somme des éléments dans un tableau $x[0..n-1]$. Analyser le temps de calcul asymptotique.*

3.2 La «taille» dépend du contexte.

En Exemple 3.1, on mesure la taille par le nombre d'éléments au lieu du nombre de bits, en supposant que les opérations numériques en Lignes 2 et 3 peuvent s'exécuter en un temps $O(1)$. Mais dans une application quelconque où on peut avoir des nombres entiers sans borne (pas seulement *int* $\leq 2^{31} - 1$ etc.), il faut considérer la dépendance du **nombre de bits utilisés dans l'encodage** de x.

Exemple 3.2. On prend l'exemple de l'**algorithme d'Euclid** qui trouve le plus grand commun diviseur de deux entiers positifs.



Le temps de calcul est **polynomiel dans la taille de l'entrée**.
Preuve : On a $a = kb + c \geq (k+1)c \geq 2c$ en Ligne E3. Donc $bc \leq ab/2$. Par conséquence, le nombre d'itérations est borné par $\lg(ab) = \lg a + \lg b$. Si $a \bmod b$ prend $O(\log^2 a)$ temps, alors l'exécution est en temps $O(\log^3 a)$. □

3.3 Récurrences

Si on travaille avec des algorithmes récursif, on obtiendra des récurrences asymptotiques.

Exemple 3.3. On prend l'exemple de la recherche de valeur maximale sur une liste chaînée. Une implantation en Java :

```
public class LinkedList // liste chaînée avec entiers
{
    private static class Noeud // liste = séquence de noeuds
    {
        private int valeur;
        private Noeud prochain;
    }

    private Noeud premier; // tete (premier element) de la liste
    // ... reste de l'implantation de la structure
```

Dans cette structure, la méthode **maxValue**() est implantée à l'aide d'une méthode auxiliaire **maxValue**(`Node` N).

```
public int maxValue()
{
  return maxValue(premier);
}

// méthode récursive auxiliare
private int maxValue(Node N)
{
  if (N == null)
    return Integer.MIN_VALUE; // la plus petite valeur possible
  else
    return Math.max(N.valeur, maxValue(N.prochain)); // récursion
}
```

Soit $T(n)$ le temps de calcul de la méthode `maxValue` quand la liste est de longueur $n \geq 0$. On a la récurrence

$$T(n) = \begin{cases} O(1) & \{n = 0\} \\ O(1) + T(n-1) & \{n > 0\} \end{cases}$$

d'où on obtient que $T(n) = O(n)$ (Exemple 2.4). □

Exemple 3.4. On prend l'exemple d'e calcul de puissances. On veut calculer x^n où $n \geq 0$ est un entier. La clé à une solution efficace est la récurrence suivante

$$x^n = \begin{cases} 1 & \{n = 0\} \\ x^{n/2} \cdot x^{n/2} & \{n > 0, n \bmod 2 = 0\} \\ x \cdot x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} & \{n > 0, n \bmod 2 = 1\} \end{cases}$$

On a donc l'algorithme

P1 **Algo** PUISSANCE(x, n)
P2 **si** $n = 0$ **alors retourner** 1
P3 **sinon**
P4 $\quad y \leftarrow$ PUISSANCE$\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)$
P5 $\quad$ **si** n est pair
P6 $\quad$ **alors retourner** y^2
P7 $\quad$ **sinon retourner** xy^2

Ceci mène à la récurrence

$$T(n) = \begin{cases} O(1) & \{n = 0\} \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(1) & \{n > 0\} \end{cases}$$

La solution est $T(n) = O(\log n)$ (Exemple 2.5).

Preuve par **substitution de variables** : par induction dans le nombre de bits dans la représentation binaire de n. Si on dénote ce nombre par b, on a la récurrence $T'(b) = T'(b-1) + O(1)$ avec la solution $T'(b) = O(b)$. Or, $b = \lceil \lg(n+1) \rceil = O(\log n)$, donc $T(n) = T'\big(O(\log n)\big) = O(\log n)$. □

Exercice 3.2. *Écrire un algorithme qui calcule x^y pour $0 \leq x, 0 \leq y < 1$. L'algorithme ne peux utiliser que les opérations arithmétiques $\sqrt{\cdot}$ (racine), multiplication, division et addition (ainsi que comparaisons $<,=,>$). Analyser le temps de calcul asymptotique en fonction du nombre de bits m dans la représentation binaire de y. (Une valeur $0 \leq y < 1$ est representée par les bits $b_1 b_2 \cdots b_m$ quand $y = \frac{b_1}{2} + \frac{b_2}{4} + \cdots + \frac{b_m}{2^m}$.)*

Exemple 3.5. Maximum par récurrence : $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(1)$, solution $T(n) = O(n)$.

Exemple 3.6. Tri par fusion : $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$, solution $T(n) = O(n \log n)$.