Chapitre 5
Méthode de programmation « Diviser pour régner »

I Introduction

I.1 Algorithme d'exponentiation rapide

Considérons un entier n et un élément x d'un ensemble muni d'une loi interne multiplicative multiplicatif $(X, *)$ (x peut donc être un nombre, une matrice, un polynôme...), et intéressons-nous au calcul de x^n, en choisissant pour mesure du coût le nombre de multiplications effectuées. Un algorithme simple et naïf vient immédiatement à l'esprit :

Code Caml 1

```
let rec puissance (x : int) (n  : int) : int =
  match n with
  | 0 -> 1
  | 1 -> x
  | n -> x * puissance x (n - 1) ;;
```

(par souci de lisibilité, tous nos algorithmes seront de type `int -> int` ; pour adapter ceux-ci à un autre type il suffira de modifier l'élément neutre 1 et le produit $*$).

À l'évidence, le nombre de multiplications effectuées est égal à $n-1$; il s'agit donc d'un algorithme de coût linéaire.

Il est cependant très facile de faire mieux, en utilisant l'algorithme suivant, connu sous le nom d'algorithme d'exponentiation rapide :

Code Caml 2

```
let rec puissance (x : int) (n : int) : int =
  match n with
  | 0 -> 1
  | 1 -> x
  | n when n mod 2 = 0 -> puissance (x * x) (n / 2)
  | n -> x * puissance (x * x) (n / 2) ;;
```

Il s'agit d'une fonction inductive dont la terminaison est justifiée par l'inégalité : $\forall n \geqslant 2, \lfloor \frac{n}{2} \rfloor < n$ et la validité par les égalités :

$$\begin{cases} x^{2k} = (x^2)^k \\ x^{2k+1} = x(x^2)^k \end{cases} \tag{1}$$

Si on note c_n le nombre de multiplications effectuées, on dispose des relations : $c_0 = c_1 = 0$, $c_{2k} = c_k + 1$ et $c_{2k+1} = c_k + 2$, soit encore : $c_n = c_{\lfloor n/2 \rfloor} + 1 + (n \mod 2)$.

Considérons la décomposition de n en base 2 : $n = [b_p, b_{p-1}, \cdots, b_0]_2 = \sum_{k=0}^{p} b_k 2^k$, avec $b_p = 1$.

Nous avons $\lfloor \frac{n}{2} \rfloor = [b_p, b_{p-1}, \cdots, b_1]_2$ et $n \mod 2 = b_0$, et il est alors facile d'obtenir : $c_n = p + \sum_{k=0}^{p-1} b_k$.

Le coût dans le meilleur des cas intervient lorsque n est une puissance de 2 : si $n = 2^p$, alors $c_n = p = \log n$ (**Dans tout le document** $\log n$ **désigne le logarithme en base 2 de** n **:** $\log n = \log_2 n = \frac{\ln n}{\ln 2}$).

Le coût dans le pire des cas intervient lorsque $n = 2^{p+1} - 1$ (l'écriture binaire de n ne comporte que des 1) : nous avons alors $c_n = 2p = 2\lfloor \log n \rfloor$.

Et il n'est guère difficile d'établir qu'en moyenne, $c_n = \frac{3}{2}\lfloor \log n \rfloor$.

Il s'agit donc d'un algorithme de coût logarithmique.

I.2 Diviser pour régner

L'algorithme d'exponentiation rapide que nous venons d'étudier est un exemple d'utilisation d'un paradigme de programmation connu sous le nom de *diviser pour régner* (ou *divide and conquer* en anglais) : il consiste à ramener la résolution d'un problème dépendant d'un entier n à la résolution de un ou plusieurs sous-problèmes identiques portant sur des entiers $n' \simeq \alpha n$ avec $\alpha < 1$ (le plus souvent, on aura $\alpha = 1/2$). Par exemple, l'algorithme d'exponentiation rapide ramène le calcul de x^n au calcul de $y^{\lfloor n/2 \rfloor}$ avec $y = x^2$.

Un autre exemple classique d'utilisation de ce paradigme est l'algorithme de recherche dichotomique : étant donné un tableau trié par ordre croissant d'entiers, comment déterminer si un élément appartient ou pas à ce tableau ?

Le principe est bien connu : on compare l'élément recherché à l'élément médian, et le cas échéant on répète la recherche dans la partie gauche ou droite du tableau.

Code Caml 3

```
let recherche_dicho x t =
  let rec aux i j =
    if j < i then false
       else match (i + j) / 2 with
             | k when x = t.(k) -> true
             | k when x < t.(k) -> aux i (k - 1)
             | k                -> aux (k + 1) j
  in aux 0 (Array.length t - 1) ;;
```

Notons c_n le nombre de comparaisons nécessaires entre x et un élément du tableau de taille n dans le pire des cas (correspondant au cas où x est supérieur à tous les éléments du tableau). Nous avons $c_0 = 0$ et $c_n = 2 + c_{\lfloor n/2 \rfloor}$ pour $n \geqslant 1$. Si on introduit de nouveau l'écriture binaire de n : $n = [b_p, b_{p-1}, \cdots, b_0]_2$, nous avons $\lfloor \frac{n}{2} \rfloor = [b_p, b_{p-1}, \cdots, b_1]_2$, et il est alors facile d'en déduire que $c_n = 2p + 2 = 2\lfloor \log n \rfloor + 2$.

I.3 Étude générale du coût

Cette partie est technique est n'est pas à connaître. Seuls les mordus s'y attaqueront ! !

En général, les algorithmes suivant le paradigme diviser pour régner partagent un problème de taille n en deux sous-problèmes de tailles respectives $\lfloor n \rfloor$ et $\lceil n \rceil$ et conduisent à une relation de récurrence de la forme :

$$c_n = ac_{\lfloor n/2 \rfloor} + bc_{\lceil n/2 \rceil} + d_n \quad \text{avec} \quad a + b \geqslant 1,$$

d_n représentant le coût du partage et de la recomposition du problème et c_n le coût total.

Nous allons faire une étude générale de ces relations de récurrence en commençant par traiter le cas où n est une puissance de 2 : posons $n = 2^p$ et $u_p = c_{2^p}$. La relation de récurrence devient : $u_p = (a + b)u_{p-1} + d_{2^p}$, soit :

$$\frac{u_p}{(a+b)^p} = \frac{u_{p-1}}{(a+b)^{p-1}} + \frac{d_{2^p}}{(a+b)^p}.$$

Par télescopage on obtient : $u_p = (a+b)^p \left(u_0 + \sum_{j=1}^{p} \frac{d_{2^j}}{(a+b)^j} \right)$.

Pour poursuivre ce calcul, il est nécessaire de préciser la valeur de d_n. Nous allons supposer que le coût de la décomposition et de la recomposition est polynomial (au sens large), ce qui permet de poser $d_n = \lambda n^k$. Nous avons alors :

$$u_p = (a+b)^p \left(u_0 + \lambda \sum_{j=1}^{p} \left(\frac{2^k}{a+b} \right)^j \right) = \begin{cases} \alpha 2^{kp} + \beta (a+b)^p \text{ si } a+b \neq 2^k \\ (u_0 + \lambda p)(a+b)^p \text{ si } a+b = 2^k \end{cases} \tag{2}$$

en ayant posé $\alpha = \lambda \left(\frac{2^k}{2^k - a - b} \right)$ et $\beta = u_0 - \lambda \left(\frac{2^k}{2^k - a - b} \right)$

Trois cas se présentent alors :

— si $a + b < 2^k$, $u_p \sim \alpha 2^{kp}$;

— si $a + b = 2^k$, $u_p \sim \lambda p (a+b)^p = \lambda p 2^{kp}$;

— si $a + b > 2^k$, $u_p \sim \beta (a+b)^p$.

Pour traiter le cas général d'un entier n quelconque, nous allons maintenant prouver le résultat suivant :

Lemme. - *Lorsque la suite $(d_n)_{n\in\mathbb{N}}$ est croissante, la suite $(c_n)_{\in\mathbb{N}}$ l'est aussi.*

Preuve. Montrons par récurrence sur $n \in \mathbb{N}^*$ que $c_n \leqslant c_{n+1}$

— Lorsque $n = 1$, on a : $c_2 = (a+b)c_1 + d_1 \geqslant c_1$ car $a + b \geqslant 1$ et $d_1 \geqslant 0$.

— Si $n > 1$, on suppose le résultat acquis jusqu'au rang $n - 1$. On a alors :

$$c_{n+1} - c_n = a(c_{\lfloor (n+1)/2\rfloor} - c_{\lfloor n/2\rfloor}) + b(c_{\lceil (n+1)/2\rceil} - c_{\lceil n/2\rceil}) + d_{n+1} - d_n \geqslant 0$$

ce qui achève la démonstration.

Lorsque cette condition est satisfaite, on peut encadrer n par deux puissances consécutives de 2 : $2^p \leqslant n < 2^{p+1}$, avec $p = \lfloor \log n\rfloor$, ce qui conduit à $u_p \leqslant c_n \leqslant u_{p+1}$. Lorsque $d_n = \lambda n^k$, les résultats précédents permettent d'établir la règle suivante (connue sous le nom de théorème maître) :

Théorème. - Lorsque $a + b \geqslant 1$, la suite $(d_n)_{n\in\mathbb{N}}$ croissante et $d_n = O(n^k)$, on a :

si $\log(a + b) < k, c_n = O(n^k)$;
si $\log(a + b) = k, c_n = O(n^k \log n)$;
si $\log(a + b) > k, c_n = O(n^{\log(a+b)})$.

I.4 Une généralisation

Dans ce qui précède la recherche a été faite en coupant en deux problème d'à peu près même taille le problème initial. On peut généraliser en le coupant en b problèmes de tailles égales (à une unité près) n/b (Attention les notations changent par rapport au théorème précédent : on note $a > 1$ le nombre total d'appels récursifs, b le nombre de sous -problèmes.)

Pour résoudre récursivement un problème, on procède donc en trois étapes :

— **Division** : diviser le problème en sous-problèmes du même type

— **Règne** : résoudre chacun des sous-problèmes par appel récursif

— **Fusion** : fusionner les résultats des sous-problèmes en la solution du problème de départ

La relation de récurrence s'écrit alors

$$c_n = ac_{n/b} + d_n + f_n,$$

d_n étant la complexité de l'étape de division, et f_n étant celle de l'étape de fusion.

On obtient alors le théorème suivant, **qui n'est pas au programme** et que nous n'utiliseront pas mais qui montre les différentes formes de complexités auxquelles nous seront confrontées dans ce contexte.

Théorème. - Lorsque $a \geqslant 1$, la suite $(d_n + f_n)_{n\in\mathbb{N}}$ croissante et $d_n + f_n = O(n^k)$, on a :

si $\log_b(a) < k, c_n = O(n^k)$;
si $\log_b(a) = k, c_n = O(n^k \log_b n)$;
si $\log_b(a) > k, c_n = O(n^{\log_b(a)})$.

En prenant $b = 2$ on retrouve le cas particulier de la première version.

Pour évaluer le coût d'une méthode suivant le principe *diviser pour régner* on étudiera plutôt à chaque fois la relation de recurrence vérifiée par le coût $c(n)$ de l'appel pour un problème de taille n. On verra également conformément au programme les cas classiques.

II Exemples

Les exemples seront présentés en cours.

II.1 Recherche dichotomique

II.2 Tri fusion

II.3 Multiplication de polynômes (Karatsuba)

II.4 Multiplication de matrices (Strassen)

II.5 Recherche du minimum d'un tableau